Borut Batagelj, Patricio Bulić,
Janez Demšar, Uroš Lotrič,
Boštjan Slivnik, Damjan Vavpotič

Učenje računalništva s pomočjo tekmovanja Bober

Skripta posodobitvenega programa nadaljnjega izobraževanja učiteljev

(Sveža elektronska verzija je dostopna na naslovu
https://ucilnica.fri.uni-lj.si/bober)

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko
2013

Damjan Vavpotič

Predstavitev podatkov

Pojem predstavitve podatkov je precej širok in ga srečamo na zelo različnih področjih (npr. predstavitev podatkov z grafikonom, besedna predstavitev podatkov, itd.) S pojmom pa se pogosto srečamo tudi pri računalništvu in informatiki, saj je ena izmed temeljnih nalog informatike prav obdelava in predstavitev podatkov, računalniki pa so danes ključno orodje za ta namen. Načini prestavitve podatkov se razlikujejo predvsem glede na to, kakšen je namen zapisa podatkov. Tako bomo podatke v primeru, da jih želimo arhivirati, predstavili na drugačen način, kot če bomo želeli tudi poiskati določen podatek izmed množice podatkov. Z bolj naprednimi načini predstavitve podatkov lahko modeliramo tudi nekatera dejstva in spoznanja iz realnega sveta. Na primer s pomočjo predstavitve podatkov v obliki drevesa lahko modeliramo organiziranost šole, s pomočjo predstavitve v obliki grafa pa meje med državami. Seveda pa je v računalništvu potrebno na nek način zapisati tudi programe oz. algoritme, ki tako predstavljene podatke obdelujejo. Če pojem predstavitve podatkov obravnavamo malce širše lahko rečemo, da moramo uporabiti ustrezno predstavitev podatkov tudi za zapis programov. Gre za povsem poseben način za predstavitev podatkov – govorimo o t.i. formalnih jezikih. Formalnih jezikov pa seveda ne uporabljamo le pri računalništvu ampak so zelo pogosto v pomoč tudi matematikom. V nadaljevanju si bomo podrobneje pogledali različna področja predstavitve podatkov. Ker pa, kot že rečeno, računalniki brez formalnih jezikov danes ne delujejo, začnimo prav z njimi.

Formalni jezik

V računalništvu (pa tudi logiki in matematiki) lahko formalni jezik opredelimo kot množico nizov simbolov (besed) omejenih s pravili. Abeceda formalnega jezika je množica simbolov iz katerih lahko sestavimo besede (nize simbolov). Poenostavljeno bi lahko tudi rekli, da je formalni jezik množica besed nad izbrano abecedo. S pojmom formalnega jezika se še posebej pogosto srečamo pri računalništvu, saj v skupino formalnih jezikov spadajo tudi vsi programski jeziki.

S perspektive poučevanja računalniškega razmišlja za otroke so principi formalnih jezikov zanimivi, ker otrokom omogočajo, da sami sestavijo preproste jezike za katere si sproti zamislijo pravila, nato pa ugotavljajo katere besede so del takšnega jezika oz. sestavljajo pravilne besede ipd.

Za lažje razumevanje zgoraj vpeljanih pojmov si poglejmo primer preprostega formalnega jezika – poimenujmo ga jezik J1. Kot abecedo jezika J1 vzemimo naslednje simbole (brez presledkov in vejic):

X , Y

Če ne opredelimo nobenih dodatnih pravil lahko z jezikom J1 sestavimo neskončno veliko besed. Poskusimo:

»X« , »Y«, »YYYYY«, »YYXYXXXYYYX«, »XXYYXXXY«, itd.

S pomočjo pravil pa lahko nabor veljavnih besed omejimo. Npr. naj velja, da je v J1 le vsak neprazen niz, ki vsebuje največ 2 simbola iz abecede. S tem pravilom smo nabor besed omejili tako močno, da jih je v jeziku J1 ostalo le še 6:

»X« , »Y« , »XX« , »YY« , »YX« , »XY«

Seveda hitro opazimo, da bi se število besed jezika J1 spremenilo tudi v primeru, da spremenimo abecedo.

Hitro si lahko zamislimo tudi bolj zapleten primer – recimo mu jezik J2. Kot abecedo jezika J2 uporabimo naslednje simbole (brez presledkov in vejic):

0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , + , =

Jezik J2 je opredeljen s pravili:

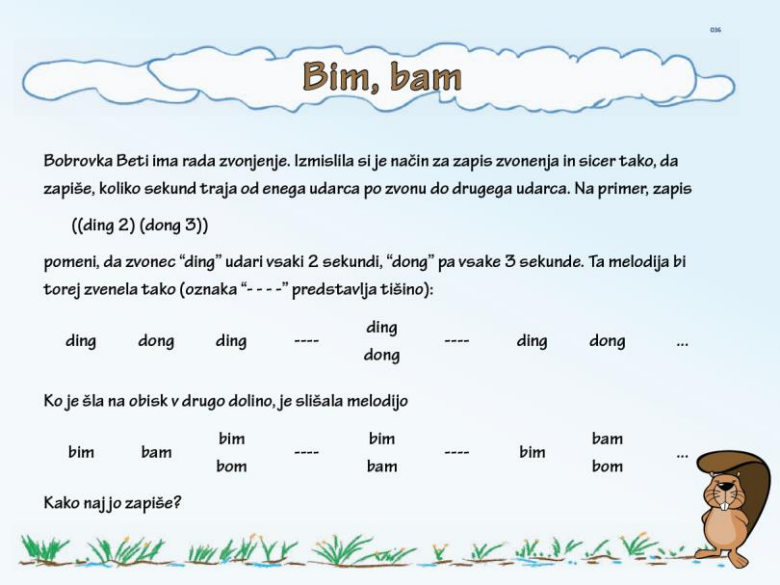
a) Vsak neprazen niz, ki ne vsebuje simbola »+« ali »=« in se ne prične z »0« je del jezika J2.
b) Niz »0« je v jeziku J2.
c) Niz, ki vsebuje »=« je v jeziku J2 izključno v primeru, da je v nizu en sam »=« in »=« ločuje dva veljavna niza iz jezika J2.
d) Niz, ki vsebuje »+« in hkrati ne vsebuje »=« je v jeziku J2 izključno v primeru, da vsak »+« v nizu ločuje dva veljavna niza iz jezika J2.
e) Noben niz razen tistih, ki jih definirajo zgornja pravila ni v jeziku J2.

Ker so pravila nekoliko bolj zapletena jih bodo otroci težje pravilno interpretirali. Po drugi strani pa gre za matematične izraze, ki so jim blizu že od prej. Vprašamo jih npr:

- Ali je niz »553+« del jezika J2? Zakaj ne? Odgovor: Ker »+« ne ločuje dveh veljavnih nizov iz J2 (pravilo d).
- In niz »2+2=4«? Seveda! Niz je skladen z vsemi pravili.
- V prejšnjem primeru bodo otroci preverjali tudi semantično pravilnost. Zato jih lahko presenetimo z nizom »1+1=4«. Če preverimo pravila, ugotovimo, da je niz skladen torej ustreza jeziku J2. Opazimo, da je s pravili definirana le sintaksa jezika (torej pravilen način zapisa), ni pa definirana semantika (pomen posameznih simbolov). Zato je del jezika J2 tudi niz »1+1=4«.

Naloge v okviru bobra se dotikajo področja na nekoliko manj formalen način. Poglejmo si primera dveh nalog.

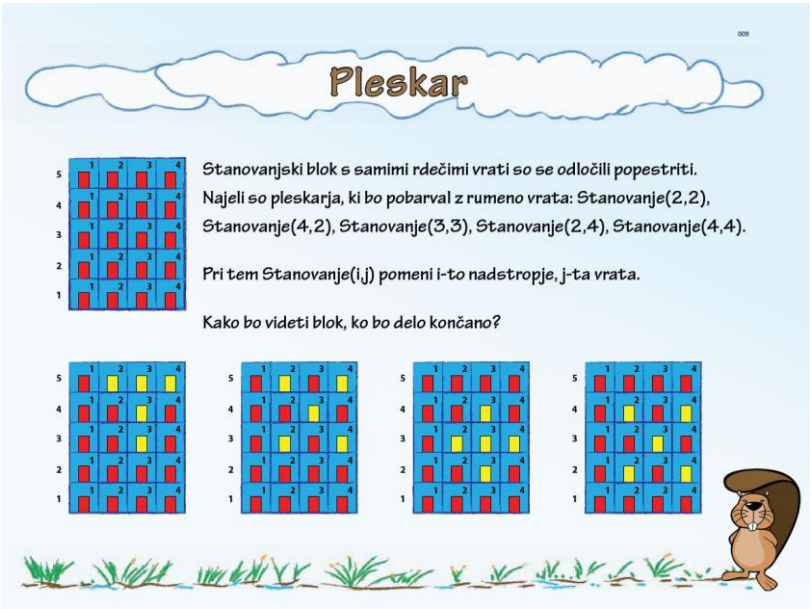
Bober - Bim, Bam



Komentar:

Pri nalogi morajo otroci zapisati melodijo bim, bam v formalnem jeziku. Čeprav so pravila jezika podana relativno ohlapno, jim je v pomoč primer ding, dong po katerem se zgledujejo in pridejo do ustreznega zapisa.

Bober – Pleskar



Komentar:

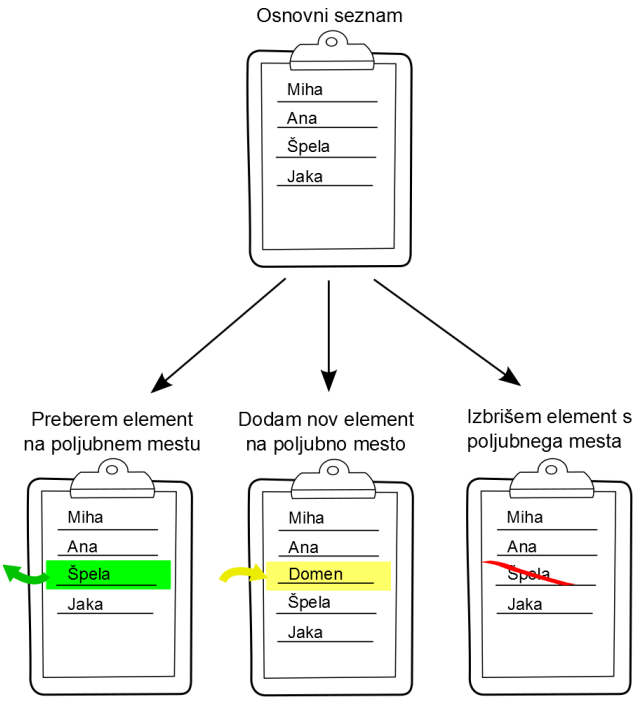
Naloga uporablja formalni jezik za zapis pobarvanih vrat. V tem primeru morajo biti otroci sposobni branja že podanega zapisa v preprostem formalnem jeziku.

Podatkovne strukture in abstraktni podatkovni tipi

V prejšnjem poglavju smo videli, da lahko s pomočjo formalnega jezika med drugim lahko predpišemo/opišemo način zapisa oz. predstavitve podatkov. Za potrebe shranjevanja in organizacije podatkov pa načinov zapisa navadno ne razvijamo povsem na novo, ampak uporabljamo različne tipične vrste podatkovnih struktur, ki podpirajo dostop do podatkov in njihovo spreminjanje. Ker nobena vrsta podatkovnih struktur ni idealna za vse namene, je pomembno poznati njihove prednosti in slabosti. Podatkovne strukture (npr. polja, zapise, datoteke, množice) dobimo z združevanjem osnovnih podatkovnih tipov. Takšne strukture podedujejo možne operacije, ki so opredeljene za osnovne podatkovne tipe, ki jih sestavljajo. To sicer programerju omogoča neovirano spreminjanje vrednosti posameznih elementov podatkovne strukture, vendar pa hkrati predstavlja veliko nevarnost pojava napak. Zato se pogosto izkaže, da je smiselno opredeliti oz. omejiti tudi vse dovoljene operacije nad elementi podatkovne strukture. V tem primeru začnemo lahko govoriti o abstraktnih podatkovnih tipih, ki hkrati predstavljajo tudi posplošitev konkretnih podatkovnih struktur. V nadaljevanju se bomo nekaterim abstraktnim podatkovnim tipom posvetili še posebej podrobno, saj se pogosto pojavljajo tudi v okviru Bobra.

Seznam

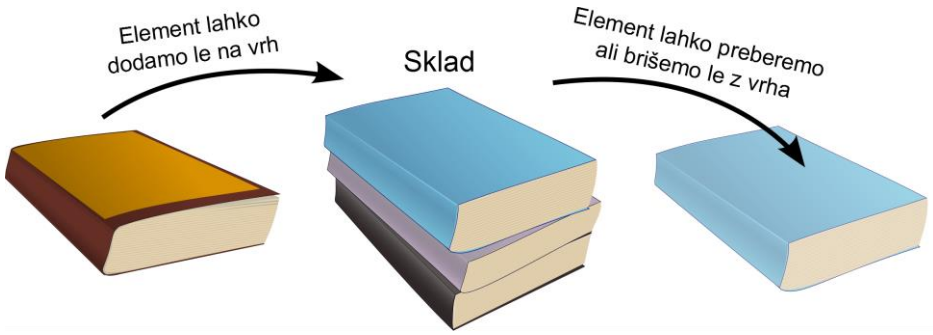
Najprej si poglejmo enega izmed najosnovnejših in hkrati najpreprostejših abstraktnih podatkovnih tipov – seznam. Seznam je zaporedje 0 ali več elementov pri katerem je vrstni red elementov pomemben, možno pa je tudi, da se elementi v seznamu ponavljajo. Operacije, ki so definirane nad seznamom nam omogočajo, da preberemo element iz poljubnega mesta na seznamu, zapišemo nov element na poljubno mesto v seznamu ali zbrišemo element na poljubnem mestu v seznamu (Slika 1).



Slika 1: Seznam

Skladi

Zdaj, ko že poznamo seznam nam sklad ne bo delal posebnih težav, saj gre le za posebno, nekoliko omejeno vrsto seznama. Pri skladu se elementi lahko berejo, dodajajo ali brišejo vedno le na začetek seznama oz. na vrh sklada. Za lažjo predstavo si zamislimo sklad knjig (Slika 2), kjer je vedno dosegljiva le prva, t.j. tista, ki je najvišje, hkrati pa novo knjigo lahko dodamo le na vrh sklada. Taki podatkovni strukturi v angleščini pravimo tudi LIFO (last-in-first-out).



Slika 2: Sklad

Poleg skladov seveda obstaja še vrsta drugih abstraktnih podatkovnih tipov. Eden zanimivejših je gotovo slovar, ki si ga bomo pogledali v sklopu naslednjega poglavja o drevesih.

Poglejmo si primer naloge iz Bobra, ki se dotika področja skladov.

Bober – Skladi krožnikov

Komentar:

Naloga na preprost način predstavi osnovno zakonitost delovanja sklada – torej, da je vedno dosegljiv le zgornji element sklada. Ker otroci tudi iz izkušenj vedo, da je krožnik iz sredine sklada težko izvleči ne da bi se kaj razbilo, naloga deluje precej intuitivno. Seveda pa lahko nalogo pogledamo tudi iz stališča vrste bobrov, ki jo je mogoče predstaviti z abstraktnim podatkovnim tipom vrsta. V tem primeru lahko ugotavljamo kakšna je razlika med strukturo LIFO (last-in-first-out) – sklad krožnikov in FIFO (first-in-first-out) – vrsta bobrov. Otroke lahko k razmišljanju pripravimo tudi z vprašanjem, ali bi bili zadovoljni, če bi na kosilo čakali skladno s pravili strukture LIFO?

Drevesa

Drevo je abstraktna struktura sestavljena iz točk oz. vozlišč (angl. *node*) in povezav med vozlišči oz. vej (angl. Branches). Vozlišča , ki nimajo podrejenih vozlišč (oz. otrok) se imenujejo listi (angl. leaf nodes). Vsako končno drevo pa ima tudi eno vozlišče, ki nima nadrejenih elementov (oz. staršev). Ta element se imenuje koren oz. korensko vozlišče (angl. root node). Posebnost drevesne strukture je, da do od vsake točke do vsake druge točke vodi le ena pot.

Drevesna struktura je tudi grafični način za predstavitev hierarhije. Z drevesnimi strukturami lahko predstavimo različne naravne ali umetne hierarhije. Npr. družinska drevesa, drevo živalskih vrst, drevo razvoja naravnih jezikov, ureditev spletne strani, struktura datotečnega sistema, struktura zaznamkov v brskalniku, itd.

Drevesne strukture pa imajo zelo pomembno vlogo tudi kot posebna oblika podatkovnih struktur. Primer abstraktnega podatkovnega tipa, ki ga pogosto realiziramo s pomočjo

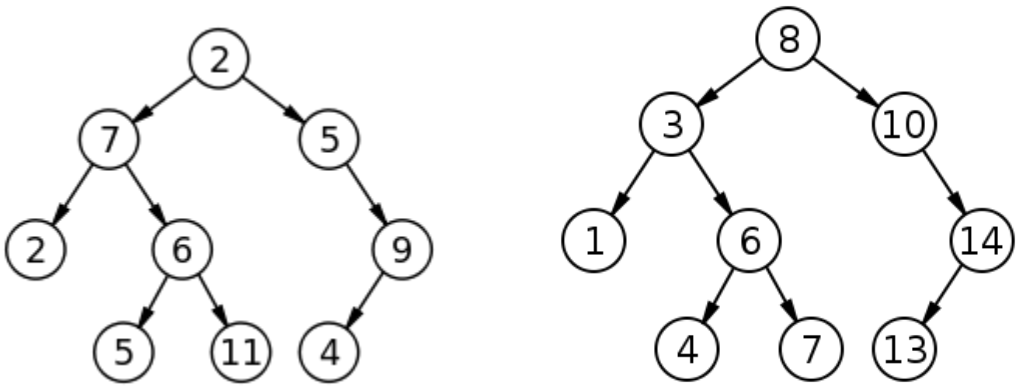
drevesa je slovar. Slovar omogoča samo tri osnovne operacije: vstavljanje, brisanje in iskanje elementa v množici. Slovar je abstraktni podatkovni tip pri katerem je pomembno čim hitrejše iskanje, brisanje in dodajanje elementov. Različne drevesne strukture, omogočajo učinkovito iskanje, brisanje in dodajanje elementov, pa tudi učinkovito iskanje minimalnega in maksimalnega elementa ter iskanje predhodnika in naslednika danega elementa. Kadar s uporabo dreves želimo optimizirati čas iskanja v slovarju nas začne zanimati poseben podtip dreves t.i. iskalna drevesa.

Preprosta drevesna struktura je dvojiško ali binarno drevo. Za to drevo je značilno, da ima vsako vozlišče največ dva otroka. Z dvojiškim drevesom lahko npr. predstavimo vse prednike neke osebe.

V računalništvu je še zlasti zanimivo binarno iskalno drevo, ki poleg lastnosti binarnega drevesa zadošča tudi naslednjim pogojem:

- levo poddrevo nekega vozlišča vsebuje le elemente z vrednostmi, ki so manjše kot je vrednost tega vozlišča,
- desno poddrevo nekega vozlišča vsebuje le elemente z vrednostmi, ki so večje kot je vrednost tega vozlišča,
- tudi levo in desno poddrevo sta binarni iskalni drevesi,
- vozlišča se ne ponavljajo.

Ključna lastnost binarnega iskalnega drevesa je, da omogoča iskanje, vstavljanje in brisanje elementov s povprečno časovno zahtevnostjo log N, pri čemer je N število vseh vozlišč (ob pogoju, da je drevo poravnano). Na spodnji sliki vidimo preprost primer binarnega iskalnega drevesa.



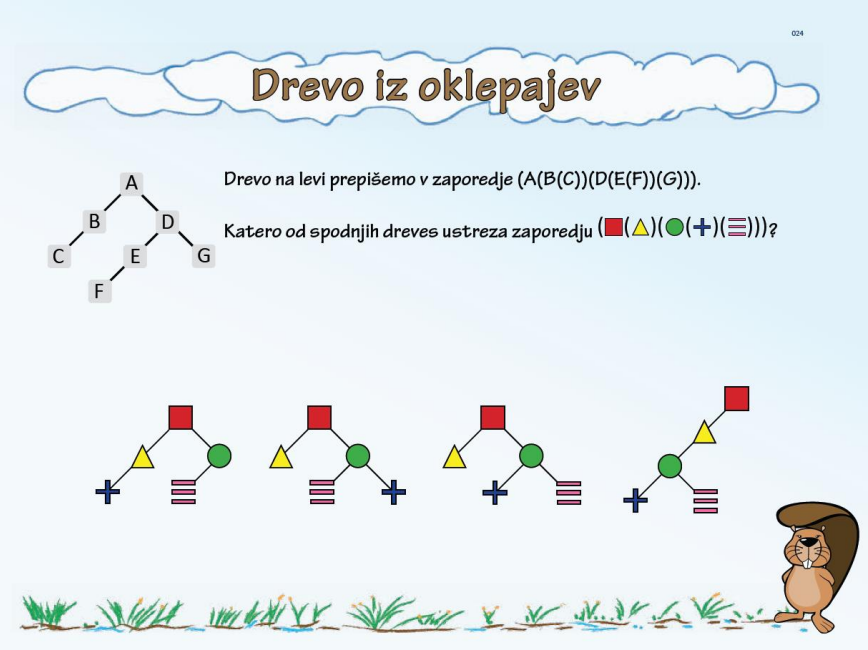
Slika 3: Binarno drevo in binarno iskalno drevo

Kot smo že omenili mora biti iskalno drevo vsaj približno poravnano, da lahko po njem učinkovito iščemo. Seveda pa se lahko v najslabšem možnem primeru binarno iskalno drevo izrodi v seznam, če npr. vanj vstavljamo elemente, ki so že urejeni po vrsti. V tem primeru je časovna zahtevnost reda N, torej enaka kot pri seznamih. Da do takšne situacije ne bi prišlo so bile izdelane še druge oblike iskalnih dreves, ki so delno in popolno poravnana (npr. lomljena drevesa, rdeče-črna drevesa, AVL-drevesa, 2-3 drevesa, B-drevesa, itd.).

Na koncu poglavja še omenimo, da lahko drevesa obravnavamo tudi kot posebno vrsto usmerjenega grafa. O tem bomo nekaj več povedali proti koncu poglavja o Grafih.

V okviru Bobra so naloge, ki se dotikajo področja dreves kar pogoste, res pa je, da imamo navadno opravka z nekoliko bolj enostavnimi primerki dreves.

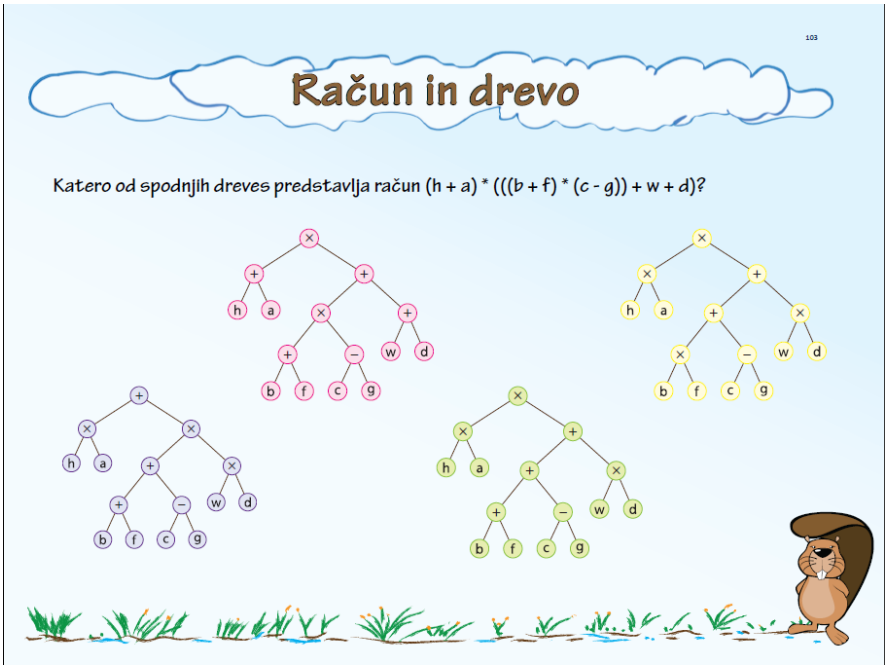
Bober – Drevo iz oklepajev



Komentar:

Drevesa je mogoče predstaviti tudi z drugačnim zapisom. Otroke lahko ob tem spomnimo tudi na formalne jezike.

Bober – Račun in drevo

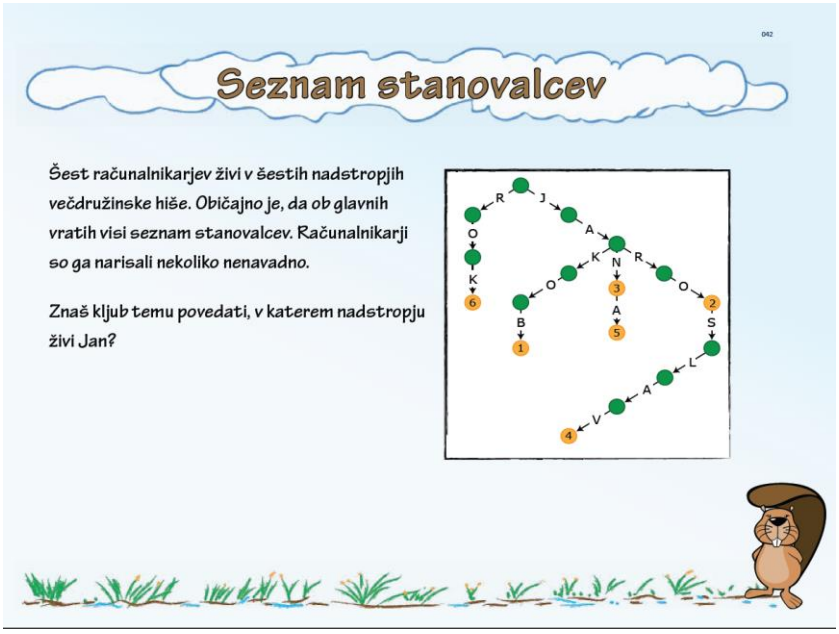


Komentar: Naloga je podobna nalogi Drevo iz oklepajev, le da je na nekoliko višji ravni.

Bober - Družinsko drevo

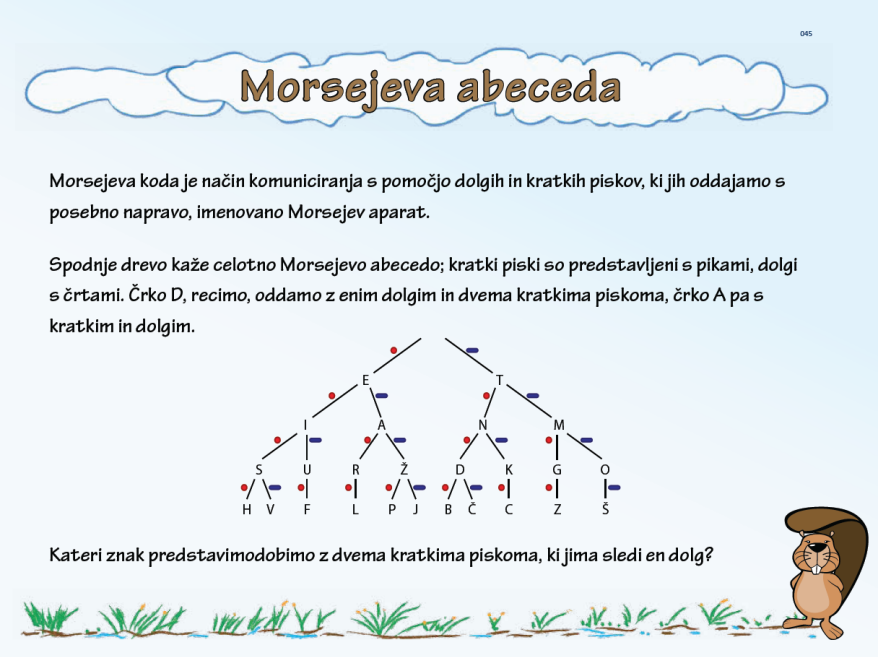
Komentar: Otroke lahko vprašamo, ali za zapis družinskega drevesa zadošča binarno drevo kjer ima lahko vsako vozlišče največ dva otroka. Kaj pa za prikaz drevesa prednikov?

Bober – seznam stanovalcev



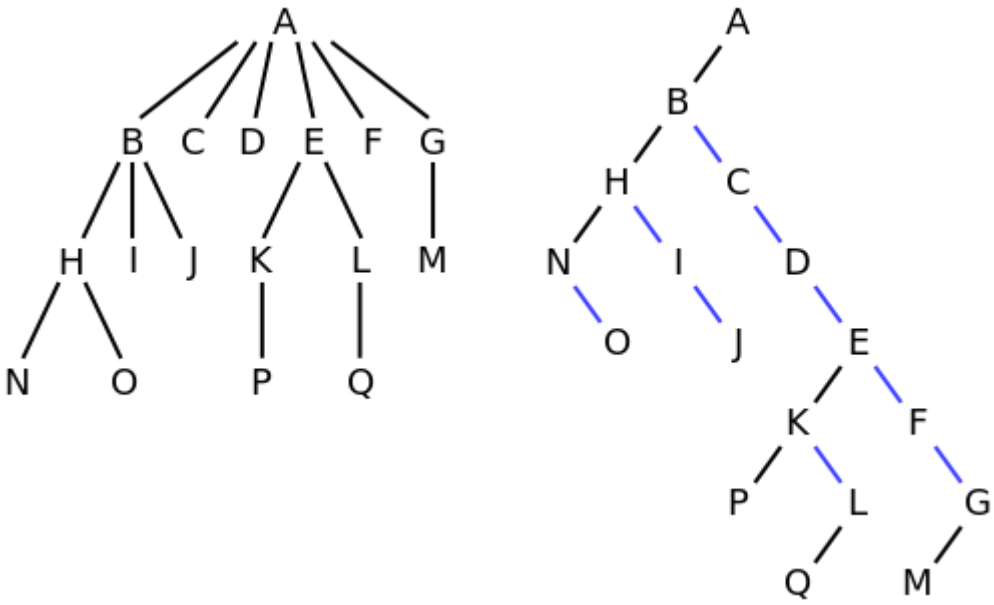
Komentar: Pri nalogi gre za nekoliko posebno drevo, saj na podlagi imen povezav prehajamo med vozlišči vse dokler ne najdemo ustreznega nadstropja. Drevo se s tem nekoliko približa ideji odločitvenih dreves (nadaljujemo po poti na kateri je prava črka), čeprav ne gre za odločitveno drevo, saj nimamo končnih dogodkov. Hkrati pa se naloga dotika tudi ideje iskalnih dreves (poiščemo nadstropje), čeprav pa hitro opazimo, da drevo ne sledi pravilom zasnove iskalnih dreves.

Bober – Morsejeva abeceda



Komentar: Naloga je zelo podobna nalogi Seznam stanovalcev, le da imajo vsa vozlišča svojo vrednost. Res pa je, da je način razmišljanja pri reševanju naloge obrnjen, saj najprej poiščemo črko – vozlišče, nato pa poiščemo pot od korena do vozlišča ter zapisujemo simbole.

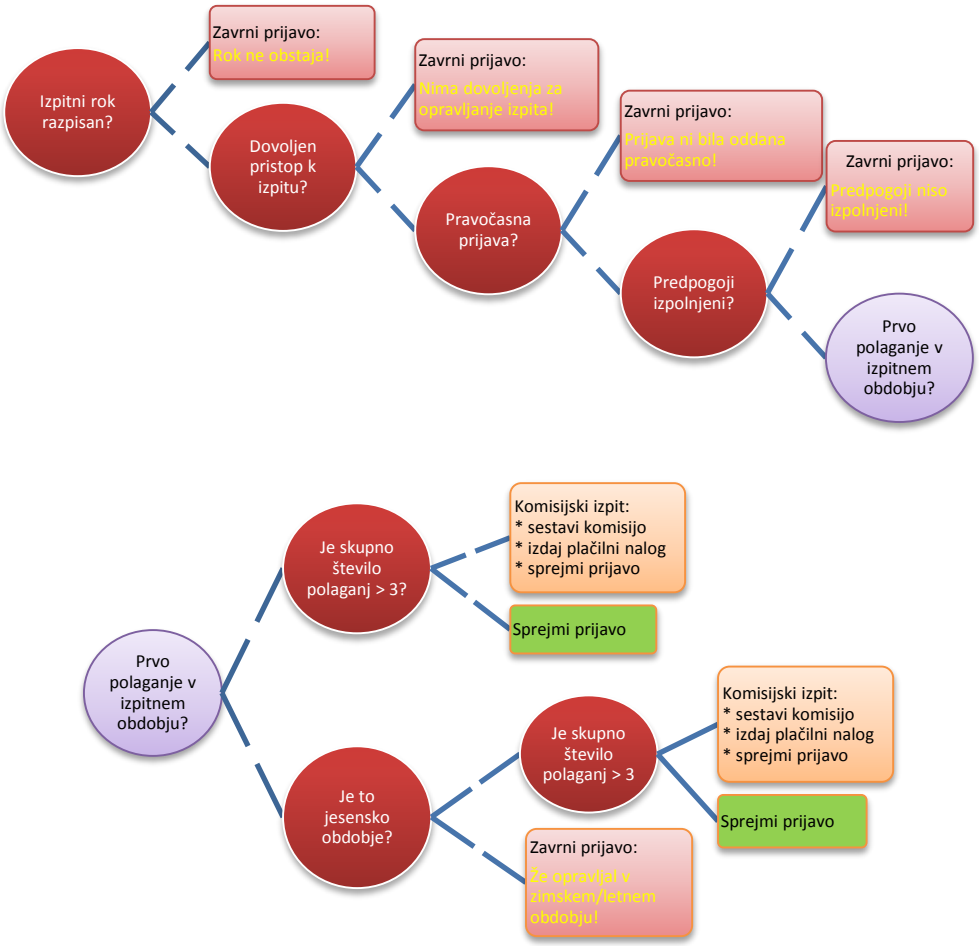
Na koncu kot zanimivost omenimo še, da lahko vsako urejeno drevo zapišemo tudi kot binarno drevo. Postopek lahko na preprost način opišemo, da moramo 2. in višje potomce nekega vozlišča povezati s prvim potomcem in sicer zaporedno (glej modre povezave). Če dobro opazujemo na nek način »zvrnemo« vse povezave med predniki in potomci razen povezave med prednikom in prvim potomcem, ki se ohrani.



Slika 4: Urejeno drevo lahko zapišemo tudi kot binarno

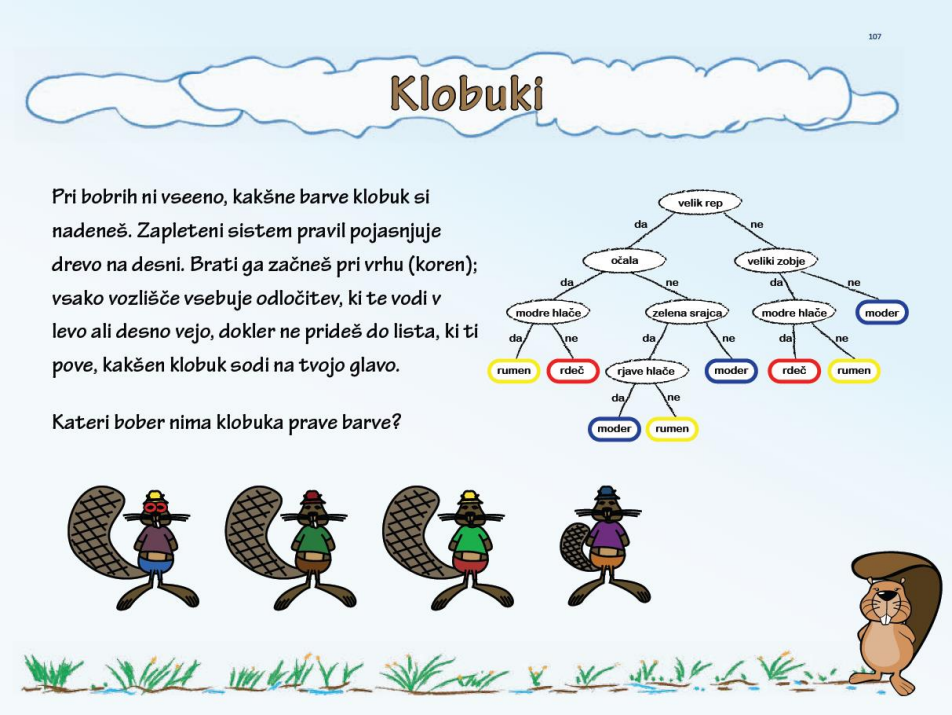
Odločitvena drevesa

Povsem posebna oblika dreves so odločitvena drevesa. Čeprav imajo podobno zgradbo kot običajna drevesa pa je njihov namen drugačen. In sicer jih uporabimo kot pomoč pri odločanju oz. pri iskanju strategije, ki nas bo najverjetneje pripeljala do želenega cilja. Uporabna so zlasti pri primerjavi različnih možnih odločitev in nam omogočajo, da razčlenimo kompleksen odločitveni problem, pri čemer lahko upoštevamo tudi različne verjetnosti dogodkov. Odločitveno drevo sestavljajo 3 različne vrste vozlišč: odločitvena vozlišča, dogodkovna vozlišča in končna vozlišča. Za odločitvenimi vozlišči sledijo povezave, ki modelirajo različne možne alternative, za dogodkovnimi pa povezave, ki modelirajo različne izide in njihove verjetnosti. Končna vozlišča ponazarjajo posledice odločitev. V primeru, da dogodkovnih vozlišč ne potrebujemo jih pri snovanju odločitvenega drevesa lahko izpustimo. Takšna odločitvena drevesa se pojavljajo tudi na Bobru. Slika 5 prikazuje primer odločitvenega drevesa brez dogodkovnih vozlišč.



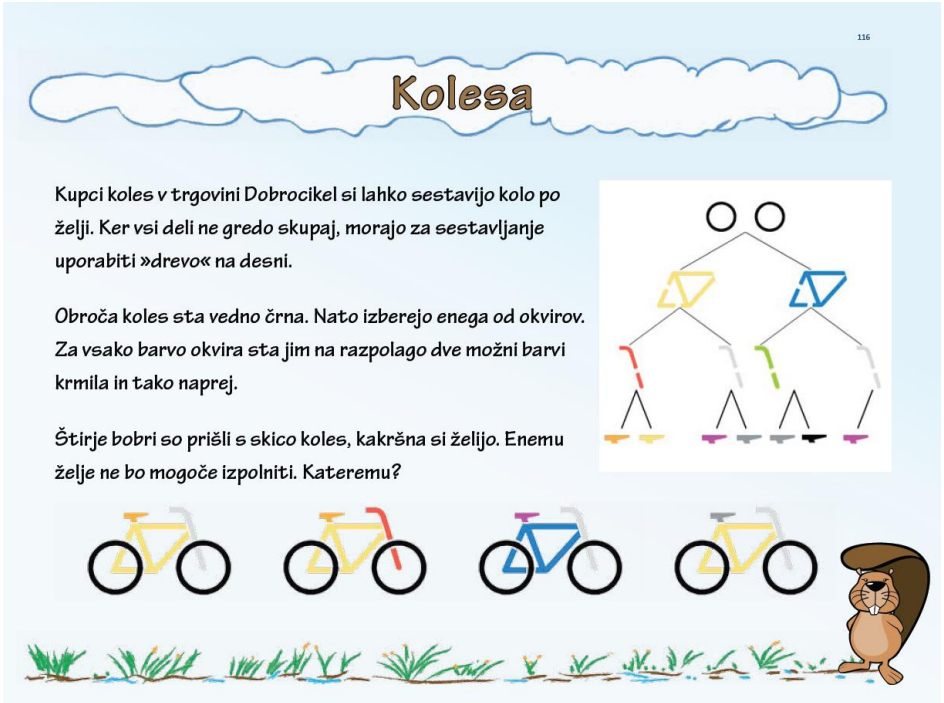
Slika 5: Primer odločitvenega modela brez dogodkovnih vozlišč

Bober – Klobuki



Komentar: Naloga prikazuje odločitveno drevo, ki ne uporablja dogodkovnih vozlišč. Na podlagi različnih lastnosti (velikost repa, nošenje očal, velikost zob in barve različnih delov obleke) otroci ugotovijo katera barva klobuka je ustrezna.

Bober - Kolesa



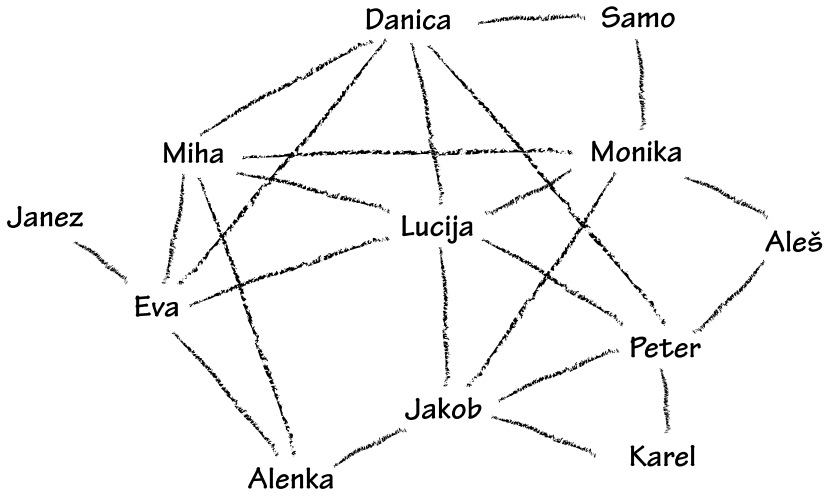
Komentar: gre za nekoliko prirejeno odločitveno drevo, saj odločitvena vozlišča hkrati predstavljajo tudi možne alternative.

Grafi

Graf je abstraktna matematična struktura, sestavljena iz "točk" (včasih jim bomo rekli tudi vozlišča; angl. *vertex* ali *node*) in "povezav" (angl. *edge*) med točkami. Z njo lahko na formalen način predstavimo najrazličnejše probleme. Točke lahko predstavljajo, na primer, osebe, povezave pa relacijo med osebami; dve osebi bosta povezani, če se poznata, sta se že rokovali na neki slavnostni večerji, imata enak vsaj en kos obleke, nimata nobenega enakega kosa obleke, sta preživeli počitnice v isti državi, govorita vsaj en skupni jezik, sta se kdaj peljali z istim vlakom, nista bili nikoli skupaj na gledališki predstavi, imata njuna vrtička skupno mejo... Točke so lahko hiše in dve hiši sta povezani, če se skozi kako okno ene hiše vidi drugo. Točke so lahko križišča in povezave ulice med njimi; ali pa kraji, povezave pa ceste med njimi. Točke so lahko avtobusne postaje in dve postaji sta povezani, če med njima vozi kak avtobus (ki med tema postajama nima vmesnih postaj). Ali pa otoki in dva otoka sta povezana, če med njima vozi neposredna trajektna povezava.

Točka je lahko povezana tudi sama s sabo, kadar je to smiselno.

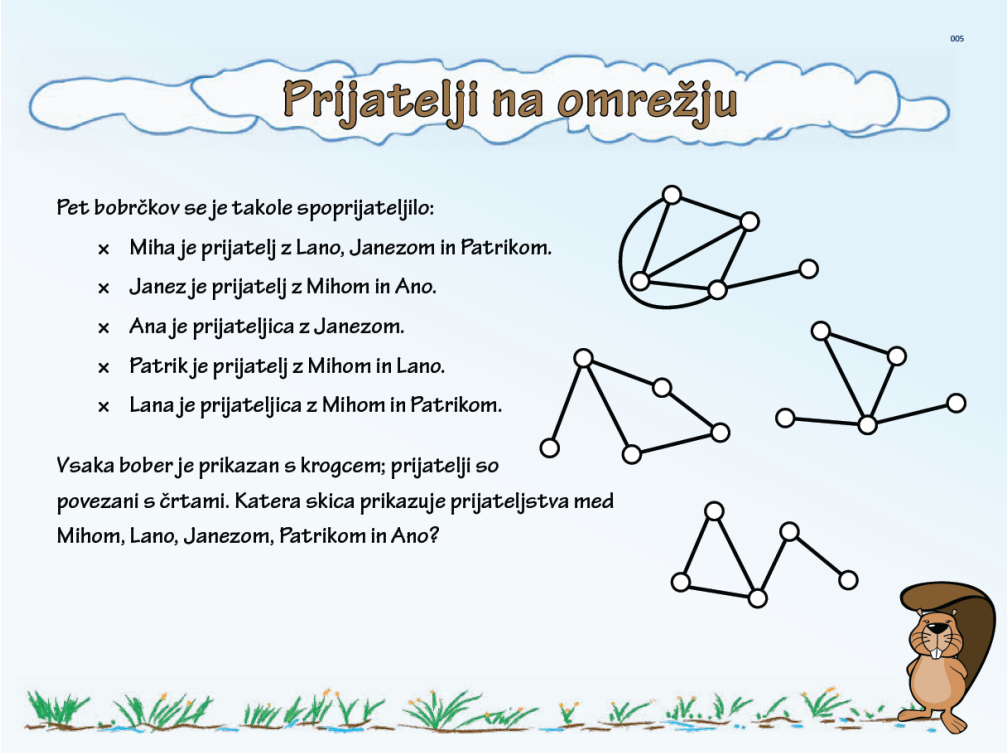
Ker vsaka točka so ustreza nekemu objektu, so točke navadno poimenovane ali kako drugače označene (Slika 6). Točkam so lahko prirejeni tudi kaki drugi podatki. Poleg imena ali oznake imajo lahko, recimo, težo, na primer velikost vrtička ali število prebivalcev kraja.



Slika 6. Primer označenega grafa: sociogram. (Vir: Bober.)

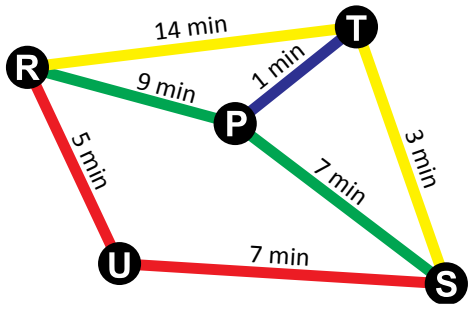
Označen graf – sociogram srečamo tudi med nalogami na Bobru.

Bober – Prijatelji na omrežju



Komentar: Naloga prikazuje več različnih grafov (sociogramov) iz katerih so bila odstranjena imena vozlišč t.j. imena prijateljev. Otroci morajo na podlagi zgradbe grafa sami ugotoviti, kateri graf prikazuje podano situacijo.

Tudi povezave imajo lahko oznake in druge podatke. Povezavam je pogosto prirejena številka, kot recimo dolžina poti med krajema, pogostost avtobusnih povezav med postajama, dolžina meje med vrtičkoma ali število voženj z vlakom, ko sta se dve osebi peljali skupaj. Primer kaže Slika 7.



Slika 7. Graf z označenimi povezavami; barva povezave pomeni avtobusno progo, številke pa povedo čas vožnje. (Vir: Bober.)

Graf je lahko usmerjen (*directed*) ali neusmerjen (*undirected*). V grafu, ki pove, ali se neka hiša vidi skozi okno druge, smer povezave pove, katero hišo vidimo iz katere. Usmerjene povezave rišemo kot puščice. Povezave, ki povedo, da dva osebi govorita skupni jezik, bodo neusmerjene.

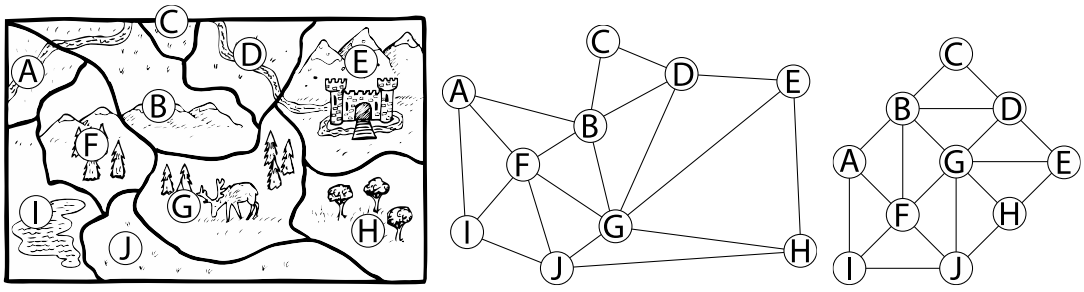
Na Bobru srečamo tudi naloge, ki se dotikajo usmerjenih grafov.

Bober - Pogrinjki

Komentar: Na podlagi sledenja vozlišč po usmerjenih povezavah grafa bodo otroci ugotovili katera miza je pripravljena skladno s pravili grafa.

Med parom točk imamo lahko eno ali več povezav. Če želimo povedati, da med dvema krajema vodi več cest, to pokažemo z več povezavami med njima. Večkratne povezave v neusmerjenih grafih sicer niso običajne, pač pa jih pogosto srečamo v usmerjenih grafih, kadar relacija med parom točk velja v obe smeri.

Ko rišemo grafe, se ne oziramo na postavitev točk: dva grafa, ki imata iste točke in povezave med njimi, sta enaka ne glede na to, kako razpostavimo točke in kje vlečemo povezave (Slika 8). Tudi, kadar graf predstavlja objekte, ki so fizično razporejeni, recimo, na zemljevidu, jih lahko rišemo v skladu s to razporeditvijo ali pa tudi ne. Pri risanju grafov gledamo predvsem na preglednosti in uporabnost.



Slika 8. Zemljevid (levo), njegova predstavitev v obliki grafa, kjer so povezane pokrajine s skupno mejo (na sredi) in pregledneje narisan isti graf (desno). (Vir: CS Unplugged in Vidra.)

Grafi so praktični, ker z njimi prevedemo najrazličnejše probleme na isti splošni, abstraktni problem. Kot primer vzemimo graf otokov in trajektov med njimi in graf oseb, v katerem povežemo tiste, ki govorijo kak skupni jezik. Tipičen problem, ki nas zanima v prvem primeru, je, ali je mogoče z določenega otoka priti na nek drug otok in kako to storiti s čim manj vožnjami ali v čim krajšem času. V drugem primeru bi se rada določena oseba pogovarjala z neko drugo, čeprav morda ne govorita nobenega skupnega jezika; zanima nas, ali obstaja "veriga" prevajalcev med njima in kako jih izbrati, da bo čim krajša. Opazimo, da gre, ko nalogo prevedemo v jezik grafov, pravzaprav za en in isti problem.

Matematiki opazujejo predvsem lastnosti grafov. V gornjem primeru vprašanje "ali obstaja?" sprašuje o lastnosti, ki ji pravimo povezanost: V računalništvu nas zanimajo tudi algoritmi na grafih. Gornji vprašanji "kako?" sprašujeta, ko ju povemo v jeziku grafov, po najkrajši poti med dvema točkama v grafu, zato ju rešujemo z enakim postopkom.

Lastnosti grafov

Matematično graf definiramo z dvema množicama: množico točk V in množico povezav E; množica povezav je podmnožica kartezičnega produkta V×V. V strogo formalno, matematične zapise v tem besedilu sicer ne bomo silili, kjer bo slovenščina dovolj jasno in nedvoumno opravila svoje delo.

Številu povezav, ki jih ima točka, rečemo stopnja točke (*vertex degree*).

Podgraf je poljubna podmnožica točk in vse povezave med njimi. Če si v grafu, ki kaže avtobusne povezave po vsej Sloveniji, izberemo le eno pokrajino ali mesto, dobimo podgraf.

Povezave, poti in obhodi

Pot (angl. *path*) med dvema točkama je zaporedje povezav, ki nas pripelje od ene točke do druge. Če so povezave usmerjene, moramo pri razmišljanju o poteh paziti tudi na smeri povezav: ko gremo od ene točke do druge, morajo biti vse povezave obrnjene v smer potovanja. Povezave in točke se lahko tudi ponavljajo; v isto točko ali po isti povezavi smemo iti večkrat.

Pri poteh nas pogosto zanima njihova dolžina. To lahko definiramo kot število povezav, iz katerih je sestavljena. Če gremo iz Ljubljane na Jesenice in od ondod v Kranjsko goro, smo napravili pot dolžine 2.

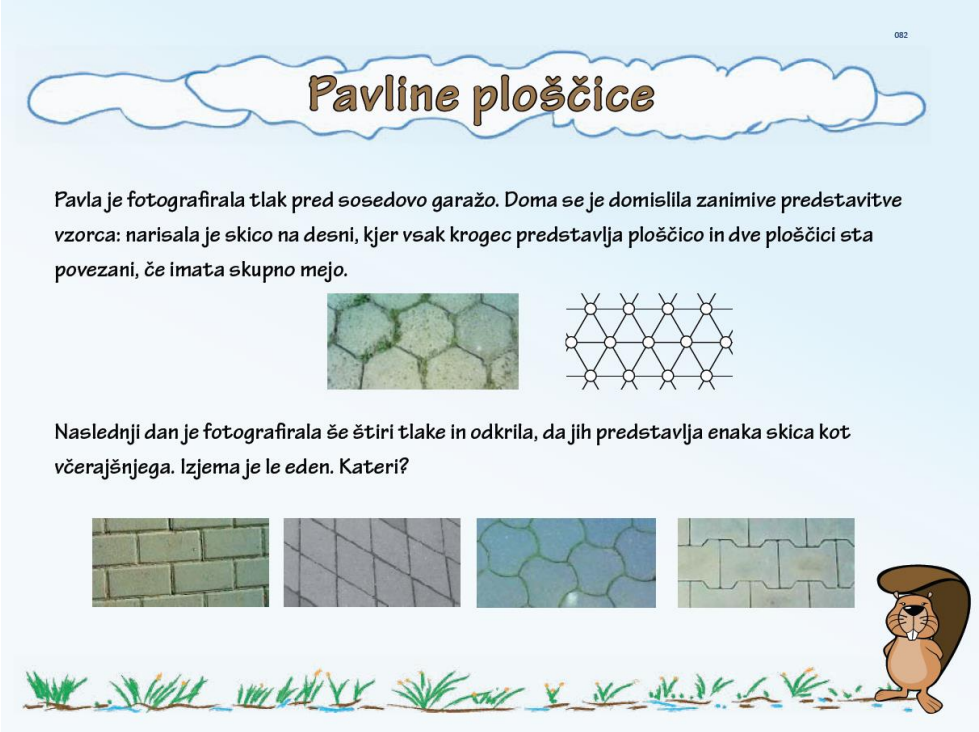
Graf je povezan (*connected*), če obstaja pot med vsakim parom točk, torej, če je iz vsake točke možno priti do vsake druge. V usmerjenih grafih govorimo tudi o šibki povezanosti (*weakly connected*): graf je le šibko povezan, če se pri kakem paru točk zgodi, da obstaja pot iz ene točke v drugo, nazaj pa ne.

Kadar graf ni povezan, razpade na nepovezane podgrafe. Rečemo jim tudi komponente. Vsaka komponenta je povezana – iz vsake točke lahko pridemo do vseh drugih točk iz komponente. Med točkami iz različnih komponent pa ni povezav. Kot primer vzemimo graf avtobusnih povezav na Hrvaškem: graf razpade na komponente, pri čemer ena, največja, predstavlja celinski del, manjše komponente pa ustrezajo posameznih otokom.

Zgodi se lahko tudi, da kakšna komponenta vsebuje eno samo točko. Takšna točka je lahko hrvaški kraj Unije, ki nima nobenih avtobusnih povezav (saj gre za edini kraj na istoimenskem otoku).

Povezavam so pogosto prirejene številke. Te imajo lahko različne pomene. V konkretnih primerih povedo, recimo, pogostost avtobusov, število skupnih jezikov ali kako dobro sogovornika govorita določen skupni jezik, kako "dobra prijatelja" sta dve osebi ali kako verjetna je določena relacija. V takšnih kontekstih temu številu rečemo *teža povezave* in v algoritmih, ki jo upoštevajo, želimo uporabiti povezave s čim večjo težo.

Bober – Pavline ploščice



Komentar: Skica, ki jo je narisala Pavla je pravzaprav graf. Da bi ugotovili, katere postavitve so ustrezne moramo biti pozorni na število skupnih mej, ki jih je Pavla predstavila s povezavami med točkami grafa.

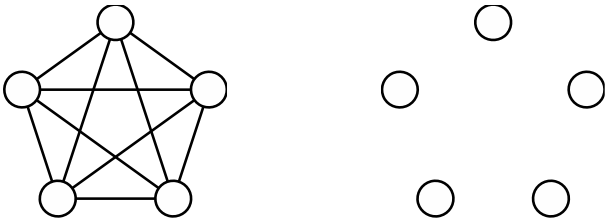
Drug pomen povezave je *cena*: v konkretnih nalogah je to lahko cena avtobusa ali trajekta ali pa dolžina poti med krajema. V algoritmih, ki upoštevajo ceno, poskušamo izbrati povezave s čim nižjo ceno. Ko iščemo, na primer, najkrajšo pot med točkama, je to pot z najmanjšo vsoto cen povezav. (V tem odstavku z "dolžino poti" očitno mislimo nekaj drugega kot malo višje; tam je bila dolžina *število* povezav, tu pa je *vsota* njihovih cen oz. dolžin.)

Če se pot konča v isti točki, kot se je začela, ji rečemo obhod (*cycle*). Med obhodi obstajata dva, ki imata posebni imeni. Hamiltonov obhod se začne in konča, kot vsi obhodi, v isti točki, prek vseh ostalih točk grafa pa gre natančno enkrat. Hamiltonova pot je podobna reč, le da se ne konča v isti točki, v kateri se je začela.

Eulerjev obhod in Eulerjeva pot sta podobna, le da ne gresta enkrat v vsako točko temveč enkrat po vsaki povezavi.

Posebni grafi

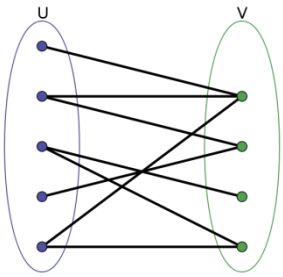
Graf je poln (*complete*) (Slika 9, levo) če so vsi pari točk neposredno povezani. Polni grafi so relativno dolgočasna zadeva. Pač pa so včasih zanimivi polni podgrafi: v (nepolnih) grafih včasih iščemo poln podgraf na petih točkah ali, v manj matematičnem jeziku, takšno peterico točk, da je vsaka neposredno povezana z vsako.



Slika 9. Poln in prazen podgraf na petih točkah.

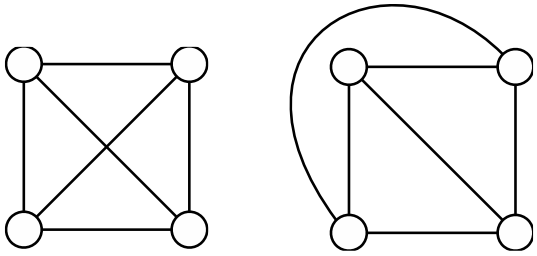
Graf je prazen, če v njem ni povezav, le točke (Slika 9, desno). Prazni grafi so absolutno dolgočasna zadeva.

Graf je dvodelen (*bipartite*), če lahko njegove točke razdelimo v dve podmnožici tako, da obstajajo povezave le med točkami iz različnih podmnožic, ne pa tudi znotraj podmnožice. Primer takšnega grafa so plesni pari: objekti so plesalci na nekem plesu in dve točki sta povezani, če sta plesalca odplesala kak ples. Dva dela tega grafa so plesalke in plesalci; povezave obstajajo le med točkami iz različnih delov (pri čemer ni potrebno, da je vsaka ženska plesala z vsakim moškim), ne pa tudi znotraj delov (ob predpostavki, da nobena ženska ni plesala z žensko in moški ne z moškim).



Slika 10: Dvodelen graf - med točkami, ki so znotraj množic U in V ni povezav

Graf je ravninski (*planar*), če ga lahko narišemo tako, da se povezave v njem ne sekajo. Vsak graf, ki ima vsaj dve povezavi, lahko seveda narišemo tudi tako, da se povezave sekajo; ravninskost pravi, da bi graf *lahko* narisali brez sekanja povezav, če bi se potrudili. Ravninskost je lastnost grafa, ne njegove grafične predstavitve.



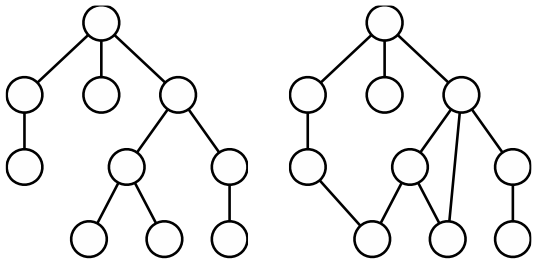
Slika 11. Graf na levi je planaren, čeprav se povezave na sliki sekajo; narišemo ga lahko namreč tudi tako, kot kaže desna slika.

Čisto posebna vrsta grafov so tudi drevesa. Tako posebna, da smo jih obravnavali ločeno: drevo je usmerjen graf, v katerem v vsako točko vodi natančno ena povezava. Izjema je le ena točka, ki nima povezav in ji pravimo koren. Drevo z *n* točkami ima, očitno, *n*-1 povezav.

Čeprav so povezave v drevesih usmerjene, jih pogosto rišemo brez puščic; pač pa drevo navadno rišemo od zgoraj navzdol (ali od spodaj navzgor ali z leve na desno), tako da se razume, da vse povezave vodijo le navzdol (ali navzgor ali na desno).

Podobna reč so usmerjeni aciklični grafi (*directed acyclic graph*; ker je ime dolgo, vrsta grafa pa pomembna, zanje uporabljamo kratico DAG, včasih pa jo uporabljamo celo kot besedo, *dag*). Zanje velja, da, kot ime pove, nimajo ciklov: iz nobene točke ne moremo priti nazaj vanjo. Spet jih lahko rišemo brez puščic, če jih, tako kot drevesa, rišemo od zgoraj navzdol.

V drevesu lahko do vsake točke pridemo le na en način, v usmerjenem acikličnem grafu pa na več načinov. Za razliko od iskanja poti v splošnih grafih, pa so poti v usmerjenih acikličnih grafih bolj "obvladljive": lažje jih je našteti ali prešteti.



Slika 12. Drevo (levo) in usmerjen aciklični graf (desno).

Bober – drevo živali

Komentar: Različne vrste oz. podvrste živali predstavimo s točkami, povezave med točkami pa predstavljajo pripadnost neke podvrste določeni vrsti. Če graf pogledamo malo bolje, lahko ugotovimo, da so povezave usmerjene (pripadnost podvrste vrsti), ter da v vsako točko vodi točno ena pot. Gre torej za drevo.

Janez Demšar

Algoritmi

Algoritem je "recept", kako rešiti nek splošen problem. Algoritem mora *točno določati postopek* – računalnik ne more ničesar delati "po občutku". Algoritem lahko uporablja naključne operacije ("izberi tri naključne elemente zaporedja"), ne more pa vsebovati operacij, za katere ni jasno, kako naj bi jih izvedli ("izberi tisti element, ki te bo najhitreje pripeljal do rešitve" – pri čemer ni jasno povedano, kateri je ta element).

Ko opazujemo algoritme, nas najprej zanima, ali so pravilni, torej, ali dajo vedno pravi rezultat. Poleg točnih algoritmov poznamo poznamo tudi hevristične: to so algoritmi, ki morda ne dajo pravega odgovora temveč samo približek. Če bi radi izvedeli, recimo, najmanjše število, ki ustreza določenim pogojem, vendar bi njegovo iskanje trajalo predolgo, zahtevalo preveč pomnilnika ali kaj podobnega, bomo navadno zadovoljni že z "približno najmanjšim" številom.

Drugo, kar nas zanima ob algoritmih, je, kako hitri so. Ker so računalniki različno hitri – in postajajo vedno hitrejši – hitrosti ne merimo absolutno, temveč relativno, glede na velikost problema. Če v tuji kuhinji iščemo predal s pokrovkami, bomo v kuhinji z dvakrat več predali potrebovali dvakrat toliko časa; algoritmom, ki se vedejo tako, pravimo, da imajo linearno časovno zahtevnost. Včasih je zahtevnost manjša od linearne in za dvakrat večji problem potrebujemo, recimo, en sam korak algoritma več. Včasih pa za dvakrat večji problem potrebujemo štirikrat toliko (in za petkrat večji petindvajsetkrat toliko) časa. So pa še hujši problemi, pri katerih že povečanje problema (na primer števila predalov) za en sam element podvoji čas, potreben za iskanje rešitve.

Algoritmi so, skupaj s podatkovnimi strukturami, eden najpomembnejših (in za mnoge najtežjih) predmetov v študiju računalništva. Kot bi vedeli povedati stari profesorji, so "o tem napisane debele knjige". V tem kratkem pregledu očitno ne bomo mogli spoznati vseh. Izbor bomo krojili po tem, kateri algoritmi se pogosto pojavljajo na tekmovanju, kateri algoritmi so najbolj poučni in kateri so najbolj zanimivi.

Predvsem algoritmi na grafih so zelo popularna tema nalog na tekmovanju Bober, gre pa le za nekaj različnih algoritmov. Če je naš namen le pripravljati otroke na tekmovanje, se tej temi gotovo splača posvetiti. Obenem so algoritmi na grafih zanimivi in privlačni, zato se jih splača poučevati tudi, če bi otrokom radi pokazali čare algoritmičnega razmišljanja. Končno, grafe v resnici srečamo na vsakem vogalu, tudi tam, kjer jih res ne bi pričakovali. Številne vsakdanje probleme – ali vsaj vsakdanje uganke – lahko prevedemo na katerega od "standardnih" problemov na grafih.

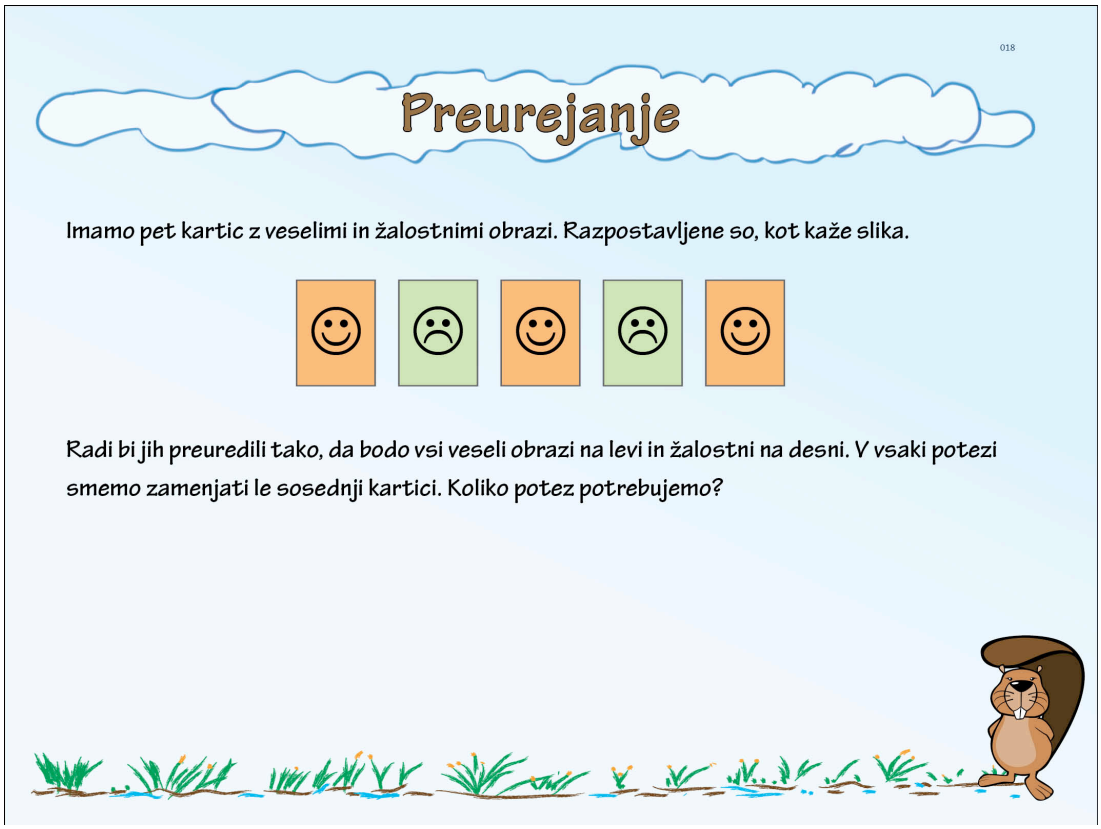
Urejanje

Algoritmi urejanja so priljubljen poligon za študij algoritmov. Po eni strani so zelo uporabni: računalniki pogosto urejajo reči po velikosti, abecedi ali kako drugače. Urejanje ni potrebno le v uporabniških vmesnikih – da pokažemo elektronsko pošto urejeno po prejemnikih, datumih ali naslovih – temveč tudi znotraj programov. Z urejanjem namreč dosežejo, da hitreje najdejo, kar iščejo; če programi ne bi interno zlagali reči na pametne načine, bi bili veliko počasnejši.

Po drugi strani so algoritmi urejanja dober primer za analiziranje "časovne zahtevnosti" algoritmov: problem je relativno preprost, predpostavke in cilji so jasne, operaciji sta le dve ("primerjaj" in "zamenjaj"), zato preživijo študenti ob analizi teh algoritmov veliko časa. Obenem je lahko analiza hitrosti delovanja teh algoritmov povsem razumljiva tudi otrokom.

S perspektive poučevanja računalniškega razmišljanja za otroke so algoritmi urejanja uporabni, ker so dovolj preprosti, da jih lahko otroci odkrijejo sami. Algoritmom urejanja sta posvečeni dve obsežni aktivnosti v okviru Vidre. Na Bobru se pojavljajo le delčki problema. Tu bomo videli nekaj nalog, nato pa spoznali nekaj algoritmov urejanja in videli, kako te naloge sodijo v širšo zgodbo.

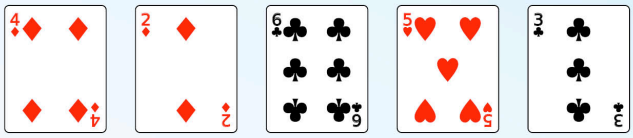
Urejanje z mehurčki



085

Urejanje kart

Spodnje karte bi rad uredil po velikosti. Pri urejanju boš vedno zamenjaval le sosednje pare kart; na začetku, na primer, lahko zamenjaš 2 in 6, ne pa, recimo, 2 in 5. Koliko zamenjav boš potreboval?



089

Postopek urejanja

Bober Donald ima nenavaden način urejanja števil po velikosti. Recimo, da mora urediti zaporedje 5, 4, 7, 2, 0, 3, 6, 1. Urejanje poteka po korakih. V prvih štirih korakih se vrstni red spreminja takole:

1. 5, 4, 7, 2, 0, 3, 6, 1
2. 4, 5, 2, 0, 3, 6, 1, 7
3. 4, 2, 0, 3, 5, 1, 6, 7
4. 2, 0, 3, 4, 1, 5, 6, 7

Kako je videti po naslednjem koraku?

- × 0, 2, 3, 1, 4, 5, 6, 7
- × 0, 1, 2, 3, 4, 5, 6, 7
- × 0, 2, 3, 4, 1, 5, 6, 7
- × 0, 2, 1, 3, 4, 5, 6, 7

Vse tri naloge so povezane z enim najpreprostejših postopkov urejanja, urejanjem z mehurčki. Kako deluje, si oglejmo kar z opisom aktivnosti z Vidre, saj ga lahko v tej obliki preprosto predstavimo tudi učencem.

1. *Izberi osem učencev, na katerih boš pokazal postopek. Postavi jih v vrsto in jim okrog vratu obesi številke v pomešanem vrstnem redu.*
2. *Razloži, da bo urejanje z mehurčki nekoliko drugačno od postopkov, ki smo jih videli doslej. Zahteva namreč več prehodov prek vrste. Učenci si bodo podajali palico: v vsakem trenutku bo palico držal en par učencev. Učenca v paru bosta primerjala svoji številki in če stojita v napačnem vrstnem redu, se zamenjata. Nato palico drži naslednji par.*
   Na primer, da so učenci razporejeni, kot kaže slika. Palico drži prvi par.

   50––20 60 30 10 80 40 70

   Ker sta obrnjena narobe, se zamenjata.

   20––50 60 30 10 80 40 70

   Nato dobi palico naslednji par.

   20 50––60 30 10 80 40 70

   Par je obrnjen pravilno, zato se ne zamenja, temveč le poda palico naslednjemu paru.

   20 50 60––30 10 80 40 70

   Ker je 60 večje od 30, se morata učenca zamenjati.

   20 50 30––60 10 80 40 70

   Nato podata palico naslednjemu paru.

   20 50 30 60––10 80 40 70

   Tako nadaljujemo. Ko pride palica do konca, je razpored takšen

   20 50 30 10 60 40 70––80

   Zadnji učenec (v gornjem primeru ta, ki ima številko 80) stopi korak vstran.

   20 50 30 10 60 40 70 80

   S tem smo končali prvi krog.

3. *Palico vrnemo prvemu paru in ponovimo vse skupaj, vendar brez zadnjega učenca – onega, ki je stopil vstran. Pride palica do konca (torej do predzadnjega učenca), stopi še ta vstran. Po drugem krogu je stanje takšno:*

20 30 10 50 40 60 70 80

4. *Palico spet dobi prvi par. Izvedemo tretji krog, ki nas pripelje do*

20 10 30 40 50 60 70 80

5. *Po četrtem krogu dobimo tole.*

10 20 30 40 50 60 70 80

Opomba: Računalnik v resnici še ne bi vedel, da je vrsta že urejena, zato bi izvedel še peti krog, v katerem pa bi opazil, da ni potrebna nobena zamenjava več in iz tega sklepal, da lahko konča z delom. Učencev s tem ni potrebno obremenjevati.

Bo rezultat algoritma vedno urejen seznam? Bo. Po prvem krogu, ko prida palica do konca, bo učenec z največjo številko prav gotovo stal na koncu vrste. Ko namreč enkrat dobi v roko palico, je ne bo več izpustil, temveč bo potoval z njo do konca. V drugem krogu bo učenec z drugo največjo številko prinesel palico do predzadnjega mesta... in tako naprej.

V tretji od gornjih nalog je uporabljen natančno takšen postopek urejanja. Naloga morda ni najbolj posrečena, saj od učencev zahteva, da postopek že poznajo in ga prepoznajo – najlažje ga prepoznamo po tem, da največja številka "roma" na desno. Če postopka še niso videli, jim ne bo lahko uganiti, kako deluje...

Prvi dve nalogi sprašujeta po nečem pomembnejšem: koliko zamenjav je potrebnih? Računalnikarje navadno zanima čas izvajanja algoritmov (ne le urejevalnih, temveč tudi drugih) v najslabšem primeru. (Zakaj v najslabšem? Morda zato, ker nas zanima, kaj nas bo čakalo v najslabšem primeru, morda pa zato, ker je iskanje najslabšega scenarija neprimerno preprosteje od izračuna poprečnega scenarija, ki si ga privoščimo le redko.)

Kaj je najslabše, kar se nam lahko zgodi pri urejanju z mehurčki? Zgodi se lahko, da bomo potrebovali le en krog manj, kot je številk. Torej, recimo, sedem krogov za osem številk. Tudi, kdaj nas to doleti, je preprosto videti: kadar je seznam obrnjen ravno narobe, 80 70 60 50 40 30 20 10. V prvem koraku bo šla 80 na desno, vse ostale številke ostanejo v takšnem vrstnem redu, kot so bile. V drugem krogu gre na desno 70, vse ostale številke spet obdržijo svoj vrstni red... in tako naprej do predzadnjega kroga, ko dobimo 20 10 30 40 50 60 70 80, da potem v zadnjem zamenjamo še 20 in 10.

Koliko zamenjav potrebujemo za to? Ko je 80 potovala na desno, je bilo za to potrebnih 7 zamenjav. Pri 70 jih je bilo potrebnih 6. Pri 60 5 ... in pri 20 1. Skupaj je to 7 + 6 + 5 + 4 + 3 + 2 + 1 = 28 zamenjav.

Za urejanje n števil po tej metodi bi potrebovali $n \times (n - 1) / 2$ zamenjav. Na Vidri je predstavljen način, kako to formulo izpeljati tudi za mlajše otroke; namesto "Gaussove" lahko uporabimo tudi drugo, jasnejšo metodo.

Za računalnikarje je $n \times (n - 1) / 2$ isto kot n^2. Razmišljamo namreč takole. $n \times (n - 1) / 2 = (n^2 - n) / 2$. Čim so številke dovolj velike, je n^2 veliko veliko večji od n, torej je $n^2 - n$ komaj kaj manj kot n^2. Ergo, $n \times (n - 1) / 2 = n^2 / 2$. Nato pa odmislimo še polovico, takole: nekateri računalniki so hitrejši, drugi počasnejši. Kar mojemu računalniku

vzame osem sekund, vzame tvojemu morda le štiri. Drugo leto pa si bom kupil štirikrat hitrejši računalnik in isti postopek bo trajal dve sekundi. Zato me bolj zanima, kaj se zgodi, če moram urejati dvakrat, trikrat ali petkrat daljše vrste števil: kolikokrat več časa bo potreboval algoritem? Zato polovico preprosto ignoriram: najsi pišem n^2 ali $n^2 / 2$, v obeh primerih bo postopek za dvakrat več števil potreboval štirikrat toliko menjav, za trikrat več devetkrat toliko in za petkrat več petindvajsetkrat toliko.

Računalnikarji rečemo, da ima algoritem kvadratno časovno zahtevnost, kar zapišemo kot $O(n^2)$. Tisti O pomeni, da smo zanemarili vse nepomembne člene (v gornjem primeru –n) in koeficiente (polovica). Za vsem skupaj je seveda še nekaj matematike (limite, konvergiranje, asimptote in podobne grozote), s katerimi pa tule raje ne strašimo.

Pri analizi algoritmov urejanja navadno ne štejemo zamenjav temveč primerjave. Pri urejanju z mehurčki razlika ni preveč pomembna, že ob naslednjem pa bomo lažje razmišljali o primerjavah.

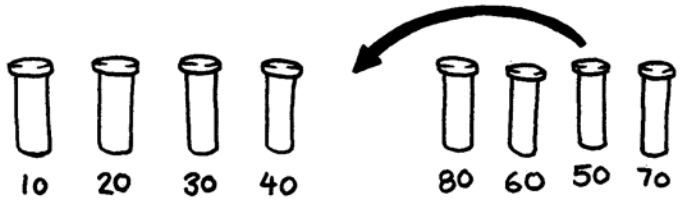
Urejanje z izbiranjem

Urejanje z izbiranjem se (kakor je videti) na Bobru še ni pojavilo, vseeno pa je prav, da vemo zanj. Gre namreč za podobno preprost postopek, ki je, poleg tega, ravno postopek, ki ga otroci tipično odkrijejo, če so prepuščeni sami sebi.

Spet poglejmo opis z Vidre, kjer otroci aktivnost izvajajo s škatlicami (na primer različno polnimi plastenkami jogurta) in preprosto tehtnico narejeno iz deščice.

Najprej poiščemo najtežjo *škatlico, tako da na tehtnico postavimo par škatlic. Odstranimo lažjo, težjo pa primerjamo z naslednjo škatlico. Spet odstranimo lažjo in vzamemo naslednjo. To ponavljamo, dokler ne preverimo vseh škatlic in ta, ki ostane, je najtežja. Postavimo jo na desno stran vrste (to je, vrste v nastajanju ;).*

Celoten postopek ponovimo na preostalih škatlicah. Ko najdemo najtežjo med njimi (torej: drugo najtežjo), jo postavimo levo od prve škatlico. Postopek spet ponovimo z ostalimi, dokler ne uredimo vseh škatlic.



Spet razmislimo ali deluje in koliko časa potrebuje.

Da deluje, je spet očitno. V vsakem koraku gotovo poiščemo najtežjo škatlico. Najtežja škatlica bo tako gotovo končala čisto na desni, druga najtežja gotovo čisto na desni od vseh ostalih in tako naprej... Rezultat mora biti urejen, drugače ne more biti.

Koliko primerjav potrebujemo? Ravno toliko kot pri mehurčkih. Ko iščemo najtežjo škatlico od osmih, bomo potrebovali sedem primerjav. Ko iščemo najtežjo od sedmih, jih potrebujemo šest ... in tako naprej. Tudi pri urejanju z izbiranjem število primerjav narašča s kvadratom števila škatlic (otrok, številk ali česarkoli že).

Hitro urejanje

Naloga spet ni posebej posrečena, saj kaže le košček algoritma, ne pa njegovega bistva. Od učencev zahteva bolj, da znajo slediti navodilom (in to ritensko), ničesar pa ne pove o algoritmu urejanja, ki se skriva za njimi.

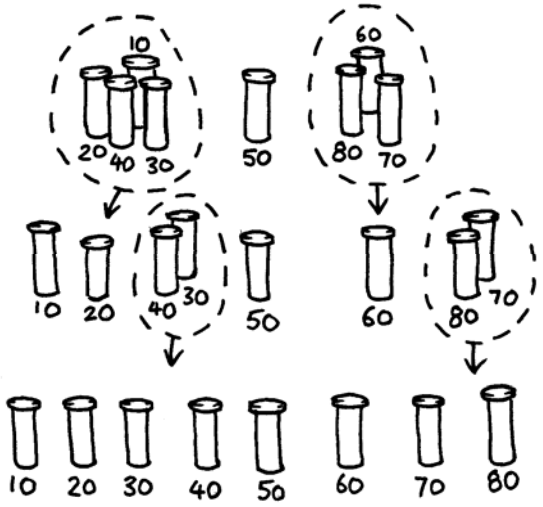
Skupaj z gornjima algoritmoma, urejanjem z mehurčki in urejanjem z izbiranjem, se pogosto predstavlja še algoritem urejanja z vstavljanjem. Tu ga bomo preskočili; dovolj je bilo. Vsem trem je skupno, da imajo kvadratno časovno zahtevnost, kar nam ni preveč všeč. A kakorkoli bi si izmišljali svoje algoritme, skoraj gotovo bi pridelali nekaj, kar ima kvadratno zahtevnost (in kaj verjetno bi se domislili le enega od običajnih treh, a v kaki preobleki).

Algoritem, o katerem (zelo prikrito) govori naloga, se imenuje hitro urejanje. To pa zato, ker je hitro. Spet si ga oglejmo z Vidro.

1. *Določi otroka, ki bo pod tvojim vodstvom urejal škatle in otroka, ki bo beležil število tehtanj.*
2. *Naključno izberi eno škatlico in jo postavi na levo stran tehtnice.*

3. *Vse škatlice primerjaj z izbrano, tako da jih eno za drugo postavljaš na desno stran tehtnice. Škatlice odlagaj na dva kupa: na levega dajaj tiste, ki so lažje in na desnega tiste, ki so težje od izbrane škatlice.*
4. *Ko si primerjal vse škatlice z izbrano, jo postavi na sredo med kupa.*

Na spodnji sliki smo si izbrali škatlico s težo 50. Lažje škatlice (10, 20, 40, 30) so na levi, težje (80, 60, 70) na desni, izbrana pa je v sredini.



Če imamo smolo, se bo pripetilo, da bo na eni strani veliko več škatlic kot na drugi; lahko se zgodi celo, da bodo vse na isti strani, ker si si izbral ravno najtežjo ali najlažjo. Nič ne de, bomo preživeli. Če škatlice niso enake, pa si zapomni, katera je približno na sredi po teži in poskrbi, da si bo otrok izbral to škatlo (lahko mu jo podaš, češ, "izberi si eno škatlo, recimo tole").

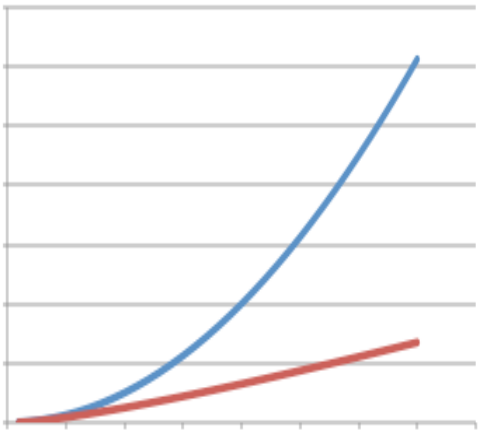
5. *Razloži otrokom, da so škatle nekako napol urejene: tista na sredi je že tam, kjer mora biti, zdaj pa moramo urediti še oba kupa. Lotili se bomo vsakega posebej. Torej:*
    a. *Med škatlicami na levi naključno izberi eno škatlico in razdeli ostale na tiste, ki so lažje in tiste, ki so težje, tako kot si to storil prej. Spet boš dobil škatlo na sredini in dva kupa, ki ju bo potrebno urediti. Lotiš se, spet, vsakega posebej...*
    b. *V desni skupini stori isto.*
6. *Kupe drobiš, dokler ne dobiš skupin z eno samo škatlico, kjer ni več kaj urejati. In, glej, škatlice so urejene!*

Gre za primer rekurzivnega algoritma: algoritem deluje tako, da razdeli problem na dva podproblema, na katerih je potrebno ponovno uporabiti prav taisti algoritem (ki bo vsakega od njiju razdelil na dva podproblema, na katerih je potrebno ponovno uporabiti prav taisti algoritem (ki bo vsakega ... in tako naprej)). Rekurzija je strah in trepet brucov, ki se učijo programiranja; kakor videvamo pri poučevanju v drugi triadi OŠ (in že pri mlajših!), pa je rekurzija v resnici tako naravna, da se otroci petega koraka zgornjih navodil, domislijo kar sami, ko jih vprašamo, kaj bomo naredili z levo in desno skupino škatlic.

Ali algoritem deluje pravilno? Da, in to tako očitno, da spet ne bomo ničesar dokazovali.

Koliko primerjav potrebuje? Tu pa postanejo stvari zanimive. Najhujše, kar se nam lahko zgodi, je, da najprej izberemo ravno najtežji (ali najlažji) element. Primerjali ga bomo z vsemi ostalimi in dobili en prazen kup in drugega, na katerem bodo vse škatle razen (ponesrečeno) izbrane. Nato se nam smola ponovi, tudi med ostalimi izberemo najtežjega (ali najlažjega). In spet in spet. Na koncu bomo potrebovali natančno toliko primerjav kot pri urejanju z izbiranjem.

Zakaj pa se potem algoritem ponaša z imenom "hitro urejanje"? Zato, ker je v poprečju najhitrejše, kar moremo narediti v okviru pravil (ki jih še ne poznamo, a jih bomo spoznali vsak čas). V poprečju potrebuje algoritem število primerjanj, ki je sorazmerno $n \ln n$. Slika kaže, kako s številom elementov naraščata funkciji n^2 in $n \ln n$.



Goljufamo? Primerjamo *poprečni* čas, ki ga potrebuje hitro urejanje, z *najslabšim* časom, ki ga potrebujejo metode, ki smo jih spoznali prej? Ne, izkaže se, da one, počasne metode, tudi v poprečju potrebujejo čas sorazmeren kvadratu števila elementov.

Pravila igre

Gre še hitreje? Si je mogoče izmisliti postopek, ki bo vedno, tudi v najslabšem primeru, potreboval število primerjav, ki bo sorazmerno $n \ln n$? Da. Takšno je, na primer, urejanje z zlivanjem.

Pa še hitreje? Postopek, ki vedno potrebuje manj kot $n \ln n$ primerjav?

To pa je odvisno od pravil igre. Običajno so takšna: števila so napisana v tabeli (ali, fizično, otroci, ki nosijo različne številke ali različno težke škatle, so postavljene na poljih). Edini operaciji, ki sta dovoljeni, sta primerjava dveh števil in zamenjava dveh elementov. Prepovedano je primerjati več števil hkrati in prepovedano je odlagati števila (elemente, škatle, otroke) v novo tabelo (ali na nova, dodatna polja). Algoritem ne sme uporabljati nobenega dodatnega prostora. Če se dogovorimo za takšna pravila, je kar preprosto dokazati, da je nemogoče razviti postopek, ki bi vedno potreboval manj kot $n \ln n$ primerjav.

Čemu ta omejujoča pravila? Prvo, da smemo primerjati le po dve števili naenkrat, izvira iz načina, na katerega so narejeni (praktično vsi) današnji računalniki. Ena od osnovnih operacij, ki jih zna izvesti "glava" računalnika, procesor, je primerjava dveh števil. Operacije, kakršno je "iskanje največjega števila med poljubno mnogimi", niso možne že zato, ker računalniški pomnilniki ne delujejo na način, ki bi to omogočal.

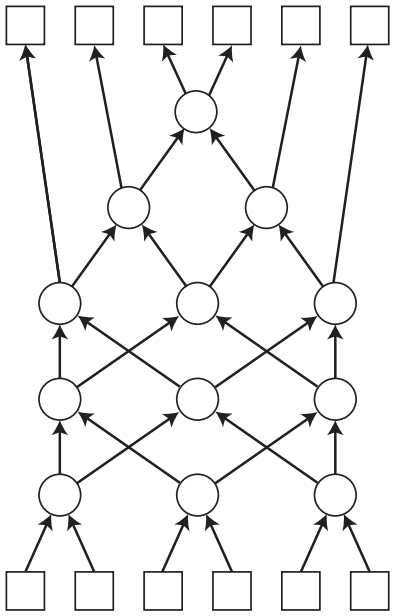
Druga omejitev izvira zgolj iz škrtosti: nočemo, da bi algoritem zapravil preveč pomnilnika. Če mu pustimo dodatni pomnilnik, lahko naredimo postopke, ki delujejo v linearnem času: dvakrat več elementov bi pomenilo le dvakrat daljši čas urejanja. Hitreje pa ne gre, saj moramo vsak element vsaj enkrat pogledati in že to nam prinese število operacij, ki je sorazmerno številu elementov.

Vzporedno urejanje

Naloga kaže postopek za vzporedno urejanje s pomočjo mreže. Kako deluje algoritem, je očitno, saj ga opisuje že naloga. Mreža iz naloge zna urediti štiri elemente. Za to sicer potrebuje pet primerjav, vendar se nekatere izvajajo vzporedno. Hitrost algoritma je takšna, kot da bi imeli le tri primerjave, vendar za to potrebujemo dve "glavi", računalnik z dvema procesorjema oz. jedroma.

Tovrstni postopki – in predvsem, kako jih predstaviti otrokom v telovadnici ali na parkirišču – je predmet posebne aktivnosti na Vidri.

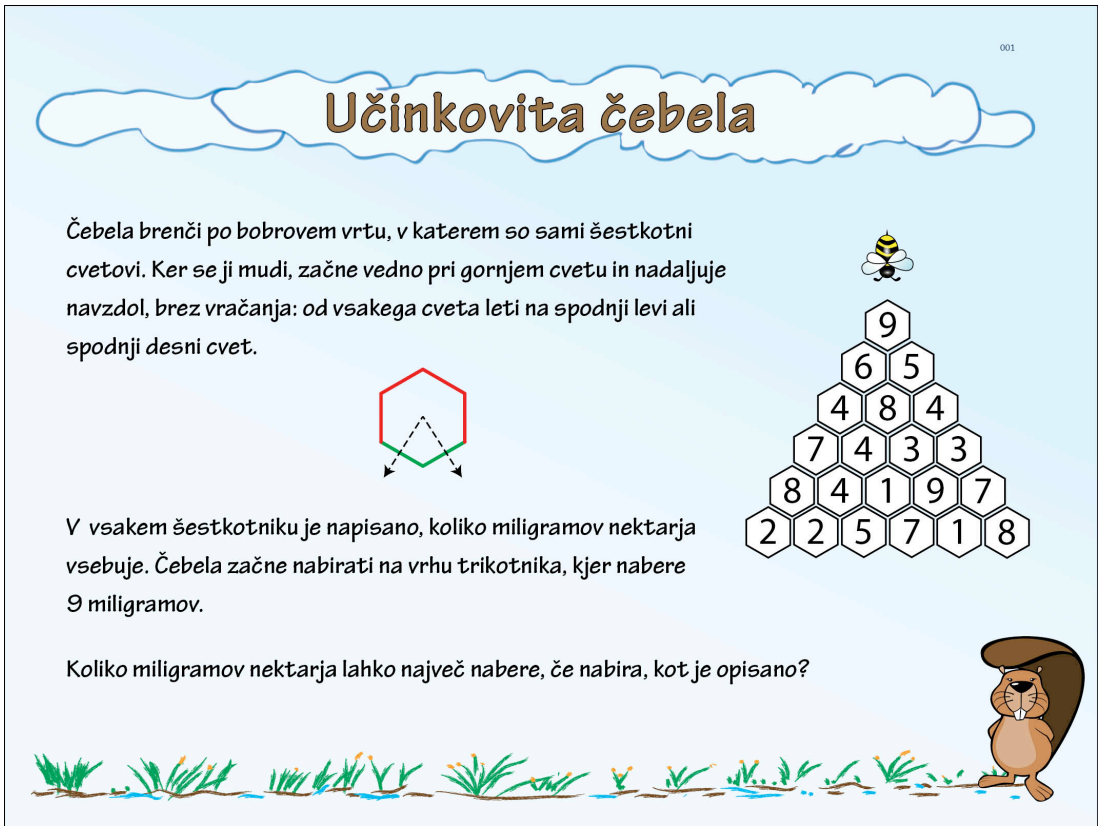
Podobne mreže lahko sestavimo tudi za večje število elementov. Tule je takšna za šest:



Hitra je tako, kot da bi imeli le pet primerjav, zahteva pa tri "glave". Kako sestavljati takšne mreže za res velike tabele, je aktivno področje raziskovanja. V praksi je dolžina mreže navadno sorazmerna številu elementov, njena širina ("število glav") pa je polovica števila elementov.

Pa imamo računalnike z "več glavami"? Tipični računalniki, ki jih kupujemo danes, imajo štiri jedra, kar pomeni, da lahko mislijo štiri misli naenkrat (to je, recimo, primerjajo štiri pare števil). Za takšne postopke urejanja to ni dovolj. Pač pa imajo dandanašnji računalniki precej zmogljive grafične kartice, ki imajo lahko tudi nekaj tisoč procesorjev. Ti so sicer povezani tako, da vsi hkrati izvajajo isto operacijo – a to je točno to, kar potrebujemo za tovrstne algoritme. V času pisanja tega besedila se grafične kartice, poleg očitnega namena, zagotavljanja čimbolj realistične grafike v igricah, uporabljajo za vzporedno procesiranje predvsem v raziskovalne namene. V času branja tega besedila pa utegne biti že drugače.

Dinamično programiranje



Dinamično programiranje ni algoritem, temveč način snovanja algoritmov. Preden ga opišemo, povejmo za dva druga.

Pristop *deli in vladaj* rešuje probleme tako, da jih razdeli na podprobleme in se loti vsakega posebej. Primer takšnega postopka je bilo hitro urejanje: namesto, da bi uredili celotno zbirko elementov, jo razdelimo na dva dela in uredimo vsako posebej. Algoritmi sestavljeni po tem vzorcu, imajo tipično tri korake: delitev, izvajanje algoritma na vsakem poddelu (ki spet obsega delitev in tako naprej) in nato združevanje. V primeru hitrega urejanja smo elemente, ki jih je bilo potrebno urediti, razdelili na manjše in večje od nekega izbranega elementa. Algoritem smo ponovili na obeh podmnožicah. Kakega posebnega združevanja pa ni bilo. Kadar delitev in združevanje zahteva čas, sorazmeren številu elementov n, bomo običajno dobili algoritme z zahtevnostjo sorazmerno $n \log n$.

Požrešni algoritmi delujejo tako, da se v vsakem koraku odločijo za trenutno najboljšo izbiro. Recimo, da smo na poti in imamo čarobni kompas, ki vedno kaže proti kraju, v katerega želimo priti. Potujemo lahko tako, da se na vsakem razpotju odločimo za pot, ki gre v najbolj pravo smer. Ali nas bo to res pripeljalo po najkrajši poti do cilja, je odvisno od tega, kako so speljane ceste.

Tudi ko s podobnim pristopom rešujemo računalniške probleme, nas pri nekaterih vrstah problemov to pripelje do najboljše rešitve, v nekaterih da slabšo rešitev od najboljše možne, v nekaterih pa nas sploh ne pripelje do rešitve. Recimo, da imamo veliko število evrskih kovancev. Plačati je potrebno določen znesek, pri čemer želimo uporabiti čim manjše število kovancev. Očitna – in pravilna, optimalna – rešitev je, da najprej odštevamo dvo evrske kovance, ko znesek pade pod dva evra, po potrebi

dodamo kovanec za evro, nato po potrebi za 50 centov, en ali dva kovanca za dvajset centov, po potrebi kovanec za deset centov, za pet centov, enega ali dva za dva centa in po potrebi kovanec za cent.

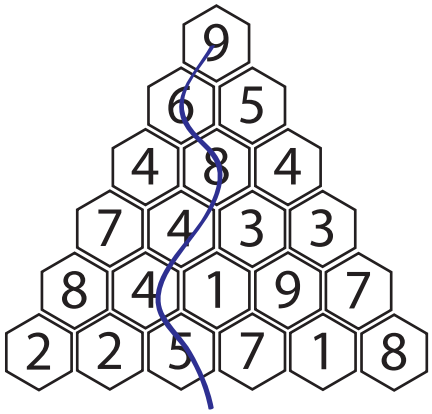
Pri kakem drugačnem sistemu kovancev postopek ne deluje. Če imamo, recimo, kovance za 10, 20, 40 in 50 centov, plačati pa je potrebno 80 centov, bi s požrešno metodo plačali s tremi kovanci (50 + 20 + 10), optimalna rešitev pa zahteva dva (40 + 40).

Še nerodneje nas lahko požrešna metoda zapelje, če imamo kovance za 25, 10 in 4 cente, plačati pa je potrebno 41 centov. Požrešna metoda izbere kovanca za 25 in 10 centov, razlike, 6 centov, pa s kovanci po 4 cente ne more plačati. Tako sploh ne najde rešitve problema – optimalne ali neoptimalne – čeprav ta obstaja (25 + 4 + 4 + 4 + 4).

Požrešne algoritme pogosto uporabljamo kot hevristične algoritme: v primerih, ko bi bilo iskanje optimalne rešitve prepočasno ali kako drugače prezahtevno, smo zadovoljni tudi s slabšo rešitvijo, ki jo najde hiter požrešen algoritem.

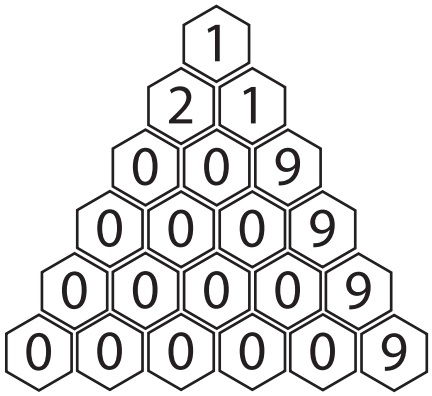
Dinamično programiranje je zanimivejša zadeva. Deluje tako, da odgovor izračuna iz rešitev preprostejših problemov. Učinkovita čebela je odličen primer tega pristopa, zato jo bomo tule temeljito secirali.

Prva – kar takoj povejmo, da ne preveč dobra – ideja, je požrešna: želva v vsakem koraku zavije k cvetu z več medu.



Rezultat je 36 mg medu. Pesimist pomisli, da je to najbrž tudi največ, kar lahko dobimo. Optimist se strinja rekoč, da je požrešna metoda gotovo dobra metoda.

Najprej razočarajmo optimista: požrešna metoda sicer lahko včasih slučajno da najboljšo rešitev, v splošnem pa ne. O tem nas hitro prepriča spodnja hudobna postavitev cvetov.



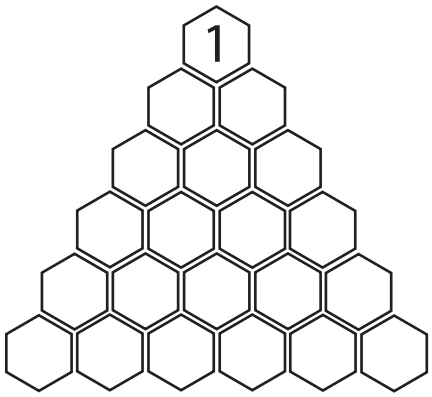
Ko čebela v prvem koraku podleže požrešnosti in odleti na cvet z dvema miligramoma medu, je njena usoda zapečatena: nabrala ne bo ničesar več. Pohlevnost v prvem koraku bi jo vodila prek bogato založenih cvetov na desni.

Tudi otroci, ki rešujejo nalogo s tekmovanja, opazijo, da bi bilo namesto zaključka 4 + 4 + 5 bolj donosno iti po poti 3 + 9 + 7 – ko bi iz cveta z 8 mg zavili na 3 namesto na 4. Tedaj se pojavijo dvomi: je 9 + 6 + 8 + 3 + 8 + 7 največ, kar lahko dobimo, ali pa bi šlo še boljše?

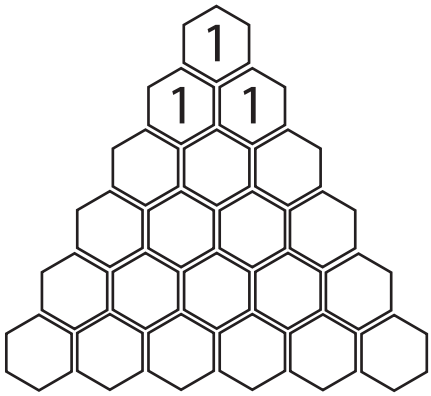
Na prvi pogled je rešitev le ena: preveriti vse možne poti. Nekateri otroci so se res lotili tega podviga; rezultat je odvisen od njihove vztrajnosti in sistematičnosti. Kdor poseduje oboje: ni težav. Nesistematični se bodo izgubili. Nevztrajni pa se bodo sredi dela vprašali: koliko tega me še čaka?

Odgovorimo jim. Na koliko različnih načinov lahko čebela preleti vrt, od gornjega polja do kateregakoli od cvetov v spodnji vrstici? Odgovor na to vprašanje nam sicer ne bo pomagalo rešiti problema (ali pač, speljal nas bo na prave misli), je pa zanimiv, zato ga le poiščimo.

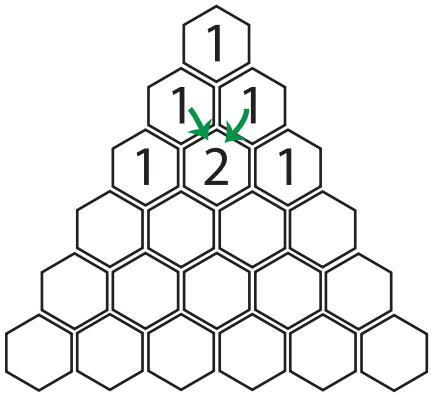
Najprej se vprašajmo nekaj preprostega: na koliko načinov lahko pridemo na prvo polje? Očitno na enega samega – tam pač začnemo.



Na koliko načinov pa pridemo na polji v drugi vrsti? Na vsakega od njiju pridemo na en sam način, namreč s prvega polja.

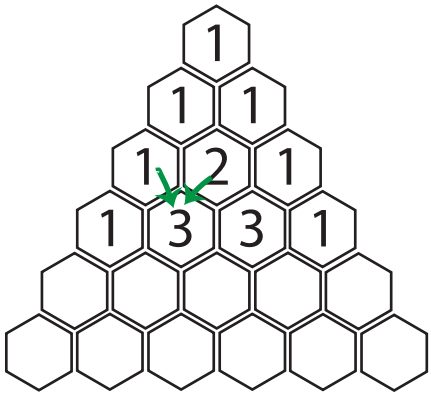


Na koliko načinov pa pridemo do polj v tretji vrsti? Na skrajno levo in skrajno desno polje pridemo na en sam način, iz skrajno levega ali skrajno desnega polja prejšnje vrste. Na srednje pa pač na dva, bodisi z leve bodisi z desne.

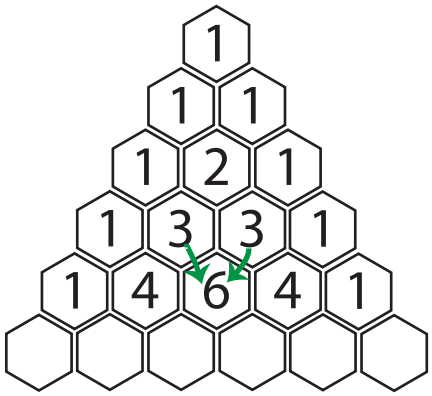


Zdaj pa postane reč zanimivejša, čeprav bo vprašanje enako: na koliko načinov pridemo z začetnega polja do posameznih polj v četrti vrsti? Na skrajni polji še vedno pridemo le iz skrajnih polj. Na drugo polje pa lahko pridemo na tri načine. Takole. Obstaja natančno en način, na katerega pridemo od začetka pa do levega polje tretje vrste; torej obstaja en način, kako priti od začetka prek levega polja tretje vrste do drugega polja četrte. Od začetnega polja do srednjega polja tretje vrste pa pridemo, kot vemo, na dva načina. Torej obstajata dva načina, da pridemo od začetnega polja prek srednjega polja tretje vrste do drugega polja četrte. Skupaj so torej 1 + 2 = 3 načini, da pridemo od začetnega polja do drugega polja četrte vrste.

Tretje polje je podobna zgodba.



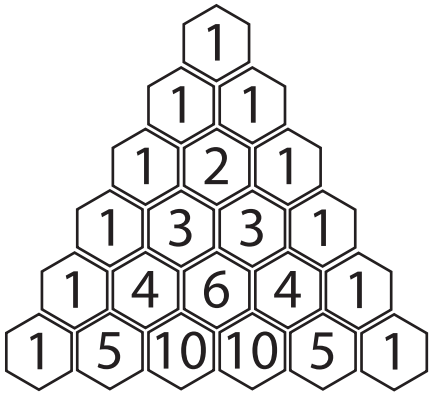
Pa četrta vrsta? Še enkrat ponovimo razmišljanje od prej, a tokrat se osredotočimo na najzanimivejše, srednje polje, tisto, v katerem piše 6.



Vanj lahko pridemo iz drugega ali tretjega polja četrte vrste. Kot že vemo, vodijo natančno tri različne poti od začetnega polja do drugega polja četrte vrste. To pomeni, da obstajajo tri poti od začetnega polja prek drugega polja četrte do srednjega polja pete vrste. Prav tako obstajajo tri poti prek tretjega polja četrte vrste. Skupaj pridemo do srednjega polja pete vrste na 3 + 3 = 6 načinov – tri poti pripeljejo z leve, tri z desne.

Podobno smo naračunali štirico: do drugega polja pete vrste pridemo na en način z leve in na tri načine z desne (ker pač obstajajo tri poti do polja na desni (ali levi, s čebeline perspektive)).

Enako naračunamo še zadnjo vrsto.



Otroci tega ne vedo, mi, ki smo odrasle, zrele osebe, pa vemo, kako se reče tej reči: Pascalov trikotnik.

Zdaj vemo, ena od možnih poti se konča na skrajnem levem polju, pet na drugem, deset na tretjem in tako naprej. Vseh možnih poti je 1 + 5 + 10 + 10 + 5 + 1 = 32.

32? Nekam čudno okrogla številka – za računalnikarja. Hm, če bi imeli eno vrsto manj, bi bilo možnih poti 1 + 4 + 6 + 4 + 1 = 16. Če bi imeli le štiri vrste, bi jih bilo 1 + 3 + 3 + 1 = 8, s tremi jih je 1 + 2 + 1 = 4, z dvema 1 + 1 = 2 in z eno 1 = 1. Nenavadno naključje?

Niti ne. Če vemo, kaj računa Pascalov trikotnik, je jasno, da mora biti tako. Pascalov trikotnik računa binomske koeficiente in z njimi velja:

$$(a+b)^n = \binom{n}{0} a^n b^0 + \binom{n}{1} a^{n-1} b^1 + \binom{n}{2} a^{n-2} b^2 + \cdots \binom{n}{n} a^0 b^n$$

če zvito rečemo $a = b = 1$, izvemo, da je $2^n = \binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n}$.

V našem primeru je *n* enak 5 (prva vrstica ima številko 0), torej imamo $2^5 = 32$ možnih poti.

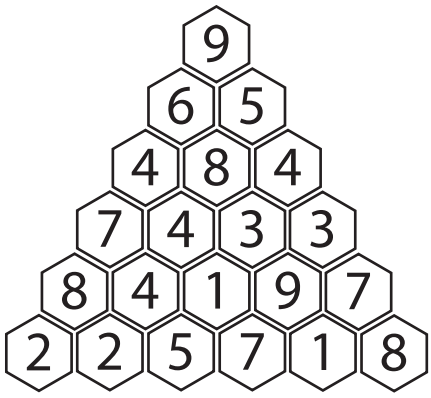
Ah, pa je bil ves ta trud res potreben? Saj gre preprosteje! Čebela se mora petkrat odločiti, kam gre. Vsakič ima dve možnosti. Z nekaj kombinatorike – dovolj preproste, da jo razložimo desetletniku, če debato začnemo z vprašanje, na koliko načinov se lahko obleče, če ima dvoje čevljev, dvoje hlač in dvoje srajc –ugotovimo, da je možnih kombinacij odločitev 2×2×2×2×2=32.

Čemu smo se mučili z vsem tem preštevanjem? Zaradi dveh stvari. Najprej, želimo se prepričati, kako narašča število poti: vsaka dodatna vrstica podvoji število poti. Algoritem, ki pravi "preveri vse možne poti" je neuporabno počasen: čas računanja ne narašča linearno s številom vrstic (dvakrat več vrstic = dvakrat več računanja), niti ne kvadratno s številom vrstic (dvakrat več vrstic = štirikrat več računanja). (Mimogrede se spomnimo, da pri algoritmih urejanja nismo marali tistih s kvadratno časovno zahtevnostjo!) Še več, ne narašča niti s kubom ali četrto potenco števila vrstic: *ena vrstica več = dvakrat več računanja*. Čas izvajanja je sorazmeren 2^n, kjer je *n* število vrstic. Računalnikarji to sorazmerje označimo z $O(2^n)$.

Takšne časovne zahtevnosti so nesprejemljive. Takšnih algoritmov ne maramo, saj delujejo le pri res res res majhnih problemih. Potrebno si bo izmisliti boljšega.

In zdaj pride drugi razlog, zakaj se nam je zdelo koristno preštevati poti: ker bomo na podoben način razmišljali, ko bomo sestavljali algoritem.

Na tem mestu bomo postali nekoliko matemetični, formalni in zateženi. Bralec, ki ga to plaši, lahko ta del mirno preskoči. S seboj povabim druge vas junake: stvar ni tako težka, zato vsaj poskusite. Z ostalimi se ponovno vidimo, ko bomo **začeli razmišljati naprej**.



Najprej se zmenimo za oznake. Polja bomo označevali tako, da bomo zapisali številko vrstice in nato številko polja. Tretje polje v šesti vrstici zapišemo s "koordinatami" (6, 3). Zanimalo nas bo, koliko medu lahko čebela največ nabere na poti do nekega polja (a, b); to število bomo označili kot $m(a, b)$. Če rečemo, recimo, m(3, 1) = 23, bo to pomenilo,

da lahko čebela na poti do (vključno) polja (3, 1) (srednje polje v tretji vrsti) nabere največ 23 mg medu.

Da bomo lažje napisali kakšno splošno formulo, recimo še, da s $q(a, b)$ označimo količino medu v polju (a, b). Tako je $q(6, 3) = 5$.

Za začetek problem rešujmo narobe, ritensko. Izberimo si neko polje, recimo tretje polje v zadnji, šesti vrsti in se vprašajmo, koliko, največ, medu lahko nabere čebela, če bo pot končala v njem, torej, koliko je m(6, 3). Odgovora sicer ne vemo, vemo pa, da čebela pride v (6, 3) bodisi s polja (5, 2) bodisi s (5, 3). Ker jo v (6, 3) čaka 5 mg medu, bo $m(6, 3)$ enak bodisi $m(5, 2) + 5$ bodisi $m(5, 3) + 5$ – toliko, kolikor nabere do enega od teh dveh polj in še 5 zraven.

Zdaj pa pazimo, $m(6, 3)$ nista dve številki, temveč ena sama: $m(6, 3)$ pove, koliko največ medu lahko čebela nabere do (vključno) polja (6, 3). Odgovor na to bi bil lahko preprost: če je m(5, 2) večje od m(5, 3), bo šla čebela, ki hoče na vsak način končati pot v polju (6, 3), raje prek (5, 2), saj bo tako nabrala več. Če je m(5, 3) večje od m(5, 2), pa bo šla do (6, 3) raje prek (5, 3). Nabrala bo torej toliko, kolikor dobi v enem ali drugem od teh dveh polj in še 5 zraven, torej $m(6, 3) = \max(m(5, 2), m(5, 3)) + 5$.

Žal pa smo mogli napisati le "bi bil lahko preprost", to pa zato, ker se nam niti ne sanja, koliko bi utegnila biti m(5, 2) in m(5, 3). A tudi na to vprašanje vemo poiskati odgovor: do (5, 2) pridemo bodisi iz (4, 1) bodisi iz (4, 2) – pač od ondod, od koder se bolj splača. V (5, 2) so 4 mg medu, zato $m(5, 2) = \max(m(4, 1), m(4, 2)) + 4$. A tu spet naletimo na isti problem: koliko medu lahko naberemo do (4, 1) in koliko do (4, 2)? Tudi ta problem rešimo na enak način.

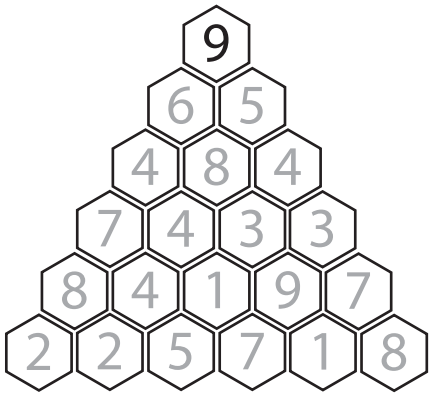
V splošnem velja: $m(a, b) = \max(m(a - 1, b - 1), m(a - 1, b)) + q(a, b)$, oziroma, na robovih, $m(a, 0) = m(a - 1, 0) + q(a, 0)$ ter $m(a, a) = m(a - 1, a - 1) + q(a, a)$.

Problem ritenskega razmišljanja je tule: ko bomo računali, koliko medu lahko naberemo do (5, 2), bomo morali izračunati, koliko ga je mogoče nabrati do (4, 2). Nekoliko kasneje se bomo vprašali, koliko medu je mogoče nabrati do (5, 3), zapisali bomo $m(5, 3) = \max(m(4, 2), m(4, 3)) + 1$ ... in ponovno računali, koliko medu je potrebno nabrati do (4, 2)!

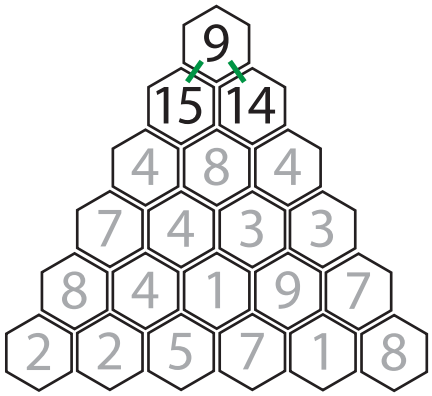
Z malo nespretnosti bomo prišli do algoritma, za katerega z malo spretnosti izračunamo, da bo potrebovali nebodijihtreba 2^n računanj. Temu se izognemo tako, da si vse, kar smo že enkrat izračunali, zapomnimo. Pri računanju nazaj je to nerodno.

Zato raje razmišljajmo naprej. (Mastni tisk je uporabljen, da pritegne vse matematikofobne bralce, za katere je besedilo od tega mesta naprej spet varno.) Razmišljanje naprej bo nenavadno podobno načinu, na katerega smo ravno prejle zabredli v računanje Pascalovega trikotnika.

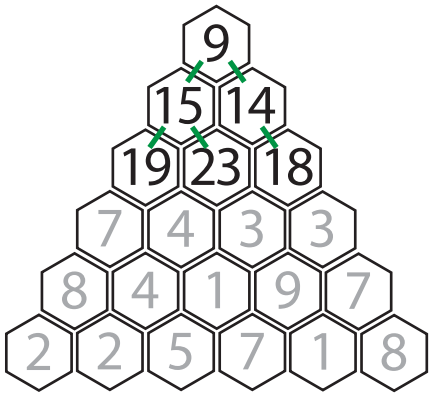
Na poti do (vključno) prvega polja čebela vedno nabere 9 mg medu.



Tudi polji v drugi vrstici ne zahtevata posebnega razmišljanja: do levega nabere 9 + 6 = 15 mg in do desnega 9 + 5 = 14 mg. Številki zapišimo kar v polje, namesto (ali prek, če rešujemo na papir) številk s količino medu.

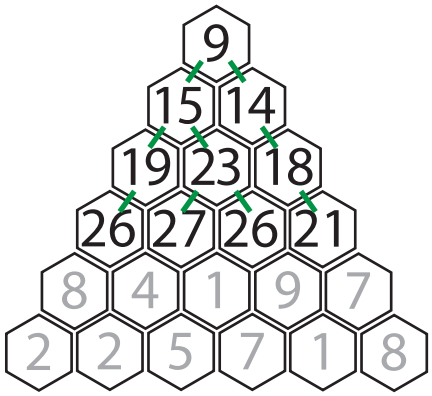


Tretja vrstica je malenkost zanimivejša:

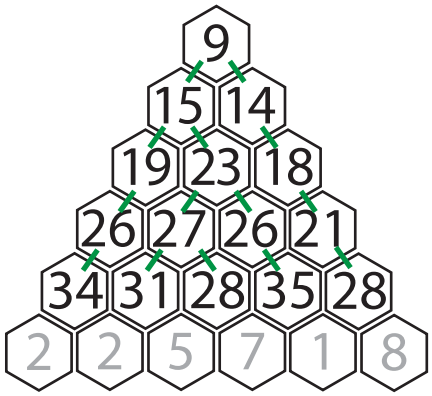


Stranski polji sta trivialni, do srednjega pa se bolj splača priti z leve, s 15. Na ta način čebela nabere 15 + 8 = 23 mg medu; če bi prišla z druge strani, bi ga le 14 + 8 = 22 mg. V polje napišemo 23 in zabeležimo, s katere strani je potrebno prileteti vanj.

V četrti vrstici skrajni polji, kot vedno, nimata dilem. V srednji dve se najbolj splača prileteti s srednjega polja tretje vrstice, saj čebela tako nabere 23 + 4 = 27 in 23 + 3 = 26 mg medu. Številki napišemo v polji in narišemo črtici, s katero označimo, odkod je potrebno priti nanju.

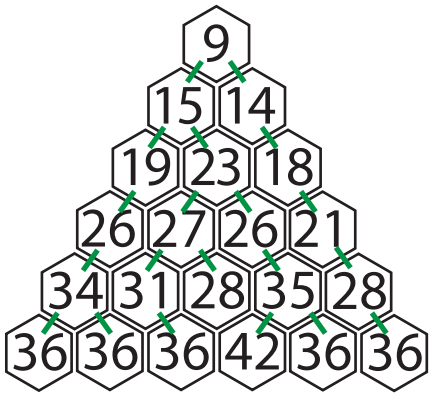


Zdaj preračunamo peto vrstico.

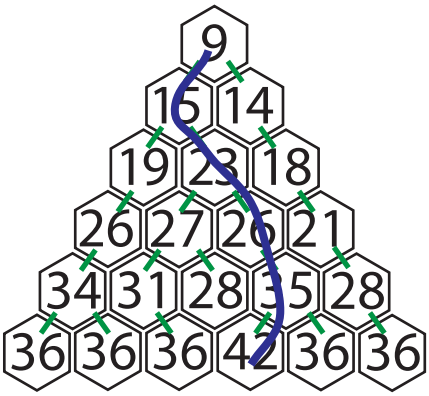


Prvo polje je jasno. V drugega pridemo z desne, s polja s številko 27 (ker je pač 27 več kot 26). Tako čebela namere 27 + 4 = 31 mg, kar napišemo v polje in dodamo črtico. Tudi v tretje polje pride z istega polja iz četrte vrstice; nabrala bo 27 + 1 = 28 mg medu, kar vpišemo in dodamo črtico. V četrto polje prileti z leve (ker je 26 več kot 21), pri čemer dobi 26 + 9 = 35 mg medu; vpišemo in narišemo črtico. Skrajno desno polje je spet trivialno.

Na koncu na enak način izračunamo še zadnjo vrstico.



Če se čebela odloči končati na četrtem cvetu zadnje vrste, lahko nabere (največ) 42 mg medu; če kjerkoli drugje, le 36. Kje pa mora iti, da nabere toliko? Zdaj sledimo poti nazaj: v ta cvet mora priti z desne (35), tja z leve (26), vanj z leve (23) in vanje spet z leve (15), vanj pa, jasno, s prvega cveta.



Koliko računov je potrebno opraviti za n vrstic? Toliko, kolikor je v n vrsticah polj. To pa je 1 + 2 + 3 + 4 + ... + n, kar je, kot že vemo, sorazmerno n^2. Ob urejanju smo nad časovnimi zahtevnostmi, ki so bile sorazmerne kvadratu števila elementov, godrnjali. Tu je to najboljše, na kar moremo upati: vsako polje moramo pač nujno pogledati vsaj enkrat, ne?

Splošno o dinamičnem programiranju

Dinamično programiranje je učinkovit in popularen pristop k snovanju algoritmov. Tule smo si izmislili algoritem za računanje optimalne poti čebele: algoritem je splošen in bi dal pravilen rezultat tudi, če bi bila količina medu v cvetovih drugačna.

Dinamično programiranje je praktično vedno, kadar lahko rešitev naračunamo iz rešitev istega problema, ki so v nekem smislu enostavnejše. Tule "enostavnost" pomeni krajšo pot: kakšna je optimalna pot do nekega cveta izvemo, če poznamo eno ali dve krajši optimalni poti. Tudi tidve naračunamo iz še krajših poti ... dokler ne pridemo do ene same poti, kjer ni več kaj razmišljati (v našem primeru do začetnega polja).

Ali, če razmišljamo naprej: zanima nas rešitev določenega problema (najboljše poti do določenega polja). Namesto, da bi rešili ta problem, rešujemo preprostejše probleme (vedno daljše poti od začetnega polja) in tako "širimo fronto" svojega znanja, dokler ne pridemo do rešitve, ki jo dejansko iščemo (količine nabranega medu v vseh končnih poljih).

Pri reševanju te naloge smo računali polja od spodaj navzgor. Fronto bi lahko peljali tudi drugače, na primer z leve proti desni – najprej bi izračunali maksimalno količino medu do vseh skrajnih desnih polj, nato do vseh drugih polj v vrsticah, pa do tretjih, četrtih... Lahko bi bili celo povsem nesistematičnih, paziti bi morali le, da za vsako polje, ki se ga lotimo računati, že poznamo količino medu v poljih nad njim.

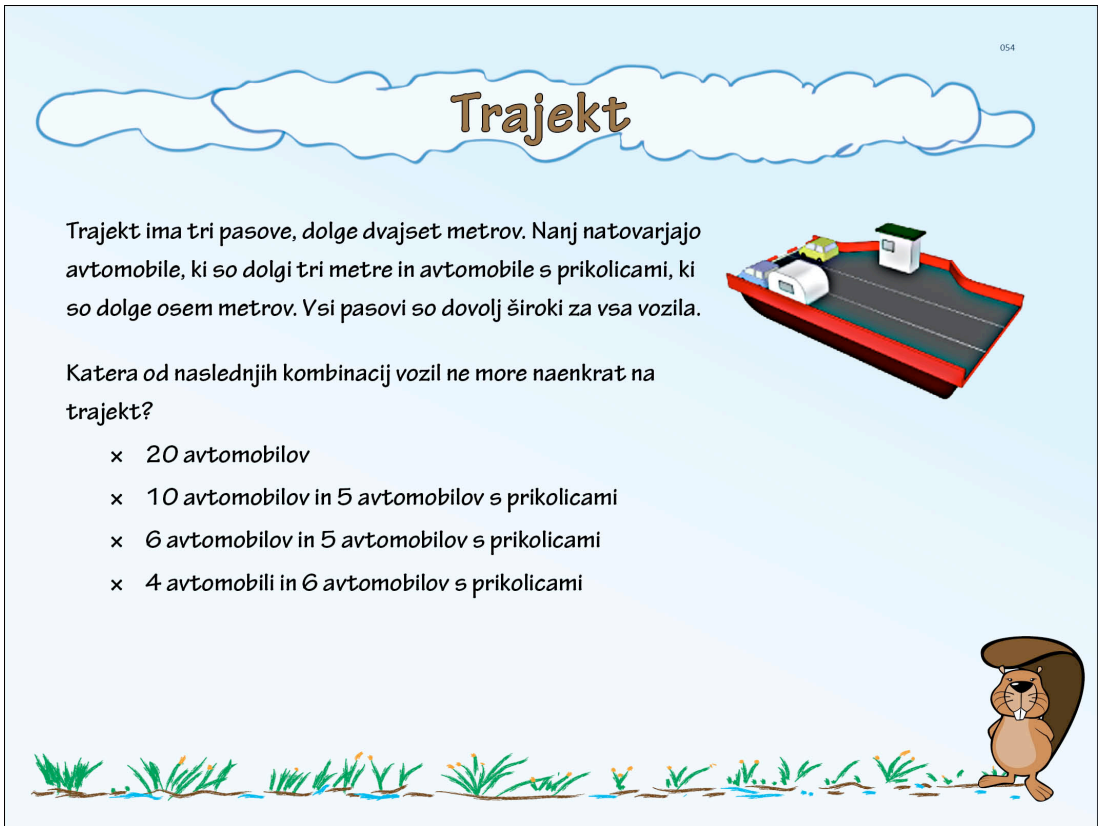
Še več, v to smo včasih prisiljeni. V nalogi s čebelo so bili cvetovi lepo zloženi v trikotnik. Dinamično programiranje pogosto uporabljamo tudi v problemih, ki nimajo tako lepe, jasne, prostorske ponazoritve. Tam niti ne moremo govoriti o "od zgoraj navzdol" ali "z leve na desno", temveč pazimo le, da gremo po vrsti v tem smislu, da vedno poznamo vse, kar je potrebno poznati za izračun vrednosti funkcije pri določenih vrednostih.

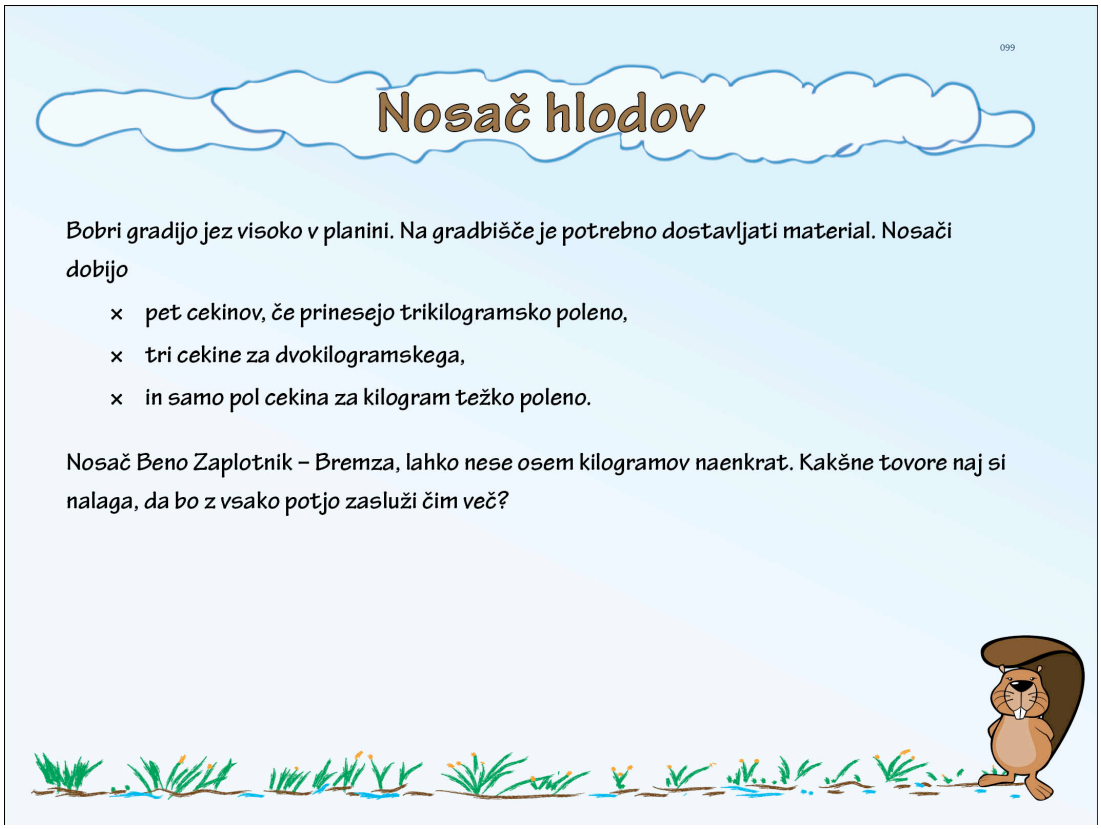
Dinamično programiranje se obnese predvsem v problemih, v katerih bi bili sicer prisiljeni večkrat računati eno in isto vrednost funkcije. Ob razmišljanju o čebeli smo

tako opazili, da bi z malo nespretnosti dvakrat računali vrednost v polju (4, 2). Dinamično programiranje nas tega reši.

V kontekstu bobra še svarilo: algoritme, sestavljene s pristopom dinamičnega programiranja (ob grafih bomo namreč spoznali še enega podobnega), je težko izvajati ročno. Naloge na tekmovanjih je zato preprosteje in varneje reševati z drugo metodo, metodo ostrega pogleda, ki pravi, da *bulji, dokler ne vidiš*. Nič pa ni narobe, če otrokom, predvsem starejšim ali bistrejšim, telovadimo možgane tudi s sistematičnim reševanjem, kakršnega smo opisali tule.

Polnjenje nahrbnika





Nalogi opisujeta enega klasičnih problemov s področja algoritmov: problem polnjenja nahrbtnika (knapsack problem). Običajna formulacija problema je takšna, kot v drugi

nalogi: imamo kup reči, za vsako je znana njena vrednost in teža. Poiskati želimo nabor z maksimalno vrednostjo, pri čemer ni dovoljeno preseči podane teže.

Obstajajo več različic problema. Prvi pravimo *0-1 nahrbtnik*: vsako reč lahko pustimo ali vzamemo, nobene reči pa ne moremo vzeti dva kosa ali več. Druga je *omejeni nahrbtnik*: vsake reči je na voljo določeno število komadov. Tretja je *neomejeni nahrbtnik*: vsake reči lahko vzamemo, kolikor je moremo nositi.

Precej lažja različica naloge je takšna, pri kateri smemo stvari rezati. Tu je stvar trivialna: pogledamo, kaj ima najboljše razmerje med ceno in težo ter natrpamo nahrbtnik z njo.

Še kar lahka različica je takšna, v kateri imajo vse reči enako vrednost glede na težo. Dva kilograma sta vedno dvakrat vrednejša od enega kilograma, pa četudi govorimo o dveh kilogramih slame in kilogramu zlata. Naloga je torej le čimbolj do roba napolniti nahrbtnik. V tej obliki nam ni neznana: vsakič, ko se odpravljamo na morje in poskušamo na različne načine stlačiti v prtljažnik vse cunje, otroške igrače, čoln in vse ostalo, rešujemo problem polnjenja nahrbtnika v teh obliki – le da je tam omejitev predvsem v obliki, ne v teži.

Pozabimo prtljažnik; recimo, da je omejitev teža. Prvi preblisk, ki ga dobimo, je požrešna metoda. Recimo, da smemo v nahrbtnik (ali, recimo raje v samokolnico) naložiti do 41 kilogramov, na voljo pa so paketi s 25, 10 in 4 kilogrami. Hm, nismo prav teh številk že nekoč videli? Jasno, požrešna metoda izbere kovanca, hočem reči paketa, po 25 in 10 kilogramov in enega za 4; tako naložimo 39 kg, v preostanek pa nimamo česa dati. Optimalna rešitev bi bila, kot se spomnimo od kovancev 25 + 4 + 4 + 4 + 4.

Požrešna metoda očitno ne bo prava. Katera pa je? Prav to je tisto: ni je. Pri metodah urejanja smo tarnali nad kvadratno časovno zahtevnostjo in se zadovoljili z n ln n. Požrešna čebela je razmislila svoje v času sorazmernem kvadratu števila vrstic (to je, v času sorazmernem številu polj). Včasih se moramo zadovoljiti s kubično časovno zahtevnostjo. Tule pa ne gre: računalnikarji verjamemo (čeprav tega (še) ne znamo dokazati), da čas, ki je potreben za reševanje problema nahrbtnika, narašča eksponentno s številom reči.

Optimalno rešitev 0-1 nahrbtnika lahko, očitno, poiščemo tako, da preskusimo vse kombinacije reči. Za tiste, ki so dovoljene (to je, niso pretežke), izračunamo njihovo skupno vrednost in si zapomnimo najboljšo. Vseh kombinacij n reči pa je ravno 2^n in vsaka dodatna reč podvoji število kombinacij.

Z omejenim nahrbtnikom je podobno: spet lahko preskusimo vse kombinacije, ki so pod dovoljeno težo, le da se zdaj ne odločamo, ali stvar dati v nahrbtnik ali ne, temveč ali jo bomo dali ničkrat, enkrat, dvakrat, trikrat in tako naprej do tolikokrat, kolikor smemo.

Kako pa ta problem rešujemo ... v resnici? Saj gre za praktičen problem, ne? Dve možnosti imamo. Nekaj se da postoriti z dinamičnim programiranjem, a to tule pustimo pri miru. Druga možnost so hevristični postopki – postopki, ki ne dajo nujno optimalnega rezultata, dajo pa "kar dobrega". Kako se hevristično lotiti nahrbtnika? Natančno tako, kot bi se ga v resničnem življenju: napolnimo, kaj odvzamemo, kaj dodamo...

Na Bobru se naloge, podobne problemu nahrbtnika, sicer kar pogosto pojavijo, vendar sestavljalci vedno poskrbijo, da je prava rešitev očitna ali pa sestavijo podatke tako, da tudi požrešna metoda da optimalno rešitev.

Algoritmi na grafih

Iskanje najkrajše poti

Med dvema točkama na grafu navadno obstaja več možnih poti. Če imajo različne povezave različno ceno, nas pogosto zanima tista, pri kateri je vsota povezav na njej najmanjša. Takšni poti rečemo *najkrajša pot*.

Algoritem, ki ga bomo spoznali, zahteva, da nobena cena ni negativna. Če ni tako, potrebujemo popolnoma drugačen algoritem, ki zahteva tudi veliko več časa (navadno celo preveč, da bi bil praktičen, zato namesto njega uporabljamo približne algoritme). Predpostavimo tudi, da iskana pot obstaja; če ciljno vozlišče sploh ni dosegljivo iz začetnega, bomo pot iskali zaman.

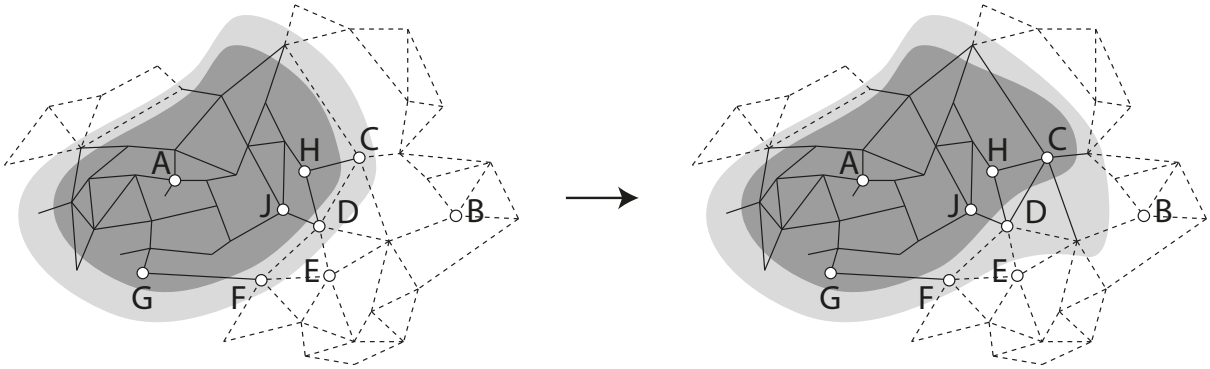
Iskanje najkrajših poti je ena najpopularnejših tipov nalog na tekmovanju Bober. Naloge na to temo: od očitnih, kjer je potrebno poiskati najkrajšo pot na zemljevidu, do prikritih, kot je iskanje najcenejšega ali najpreprostejšega zaporedja del, ki nas pripelje do končnega izdelka.

V nalogah, s kakršnimi se spopadajo osnovnošolci na tekmovanjih, je najkrajšo pot navadno mogoče poiskati kar z metodo ostrega pogleda. Sestavljene so namreč tako, da je število poti (razen očitno predolgih, na primer takšnih, v katerih se točke ponavljajo) dovolj majhno, da jih lahko sistematično pregledamo, ali pa se različne poti na enem ali več mestih združijo in lahko nalogo rešujemo tako, da iščemo najkrajše poti po kosih, med posameznimi "ozkimi grli".

Računalnik nima ostrega pogleda, pa tudi ljudje pri dovolj velikih grafih ne moremo več biti prepričani, da smo res preverili vse možnosti. Tedaj uporabimo algoritem, ki je dobil ime po slavnem nizozemskem računalnikarju Edsgerju W. Dijkstri: Dijsktrin algoritem.

Algoritem ne poišče le najkrajše poti od začetne do ciljne točke, temveč tudi najkrajše poti do množice drugih točk. To počne tako, da začne pri začetni točki in nato postopno širi množico točk, do katerih pozna najkrajšo možno pot.

Točkam, do katerih je že odkrita najkrajša pot, bomo – iz razlogov, ki bodo postali jasni vsak čas – rekli *obiskane točke*. Točke, ki so neposredno povezane z obiskanimi, rečemo *mejne točke*. Algoritem si bomo najprej ogledali na skici, nato še na konkretnem primeru.



Poiskati želimo najkrajšo pot od A do B na gornji sliki. Začnimo na levi strani. Temnejši del kaže obiskane točke. Zanje že vemo, kakšna je najkrajša pot od A do njih – prek katerih točk vodi in koliko je dolga.

Mejne točke so v svetlejšem delu; primeri takšnih točk so C, D in F. Za mejne točke še ne poznamo najkrajše poti. Seveda lahko izračunamo, kako dolga bi bila, recimo, najkrajša pot do F, ki bi vodila prek G: k (že znani) dolžini najkrajše poti do G prištejemo dolžino povezave med G in F. Prav tako lahko za D izračunamo dolžino poti prek H (kot vsoto znane najkrajše poti do H in dolžine povezave med H in D) ter prek J; najkrajša pot do D, ki vodi naposredno iz obiskanega dela, je dolžina krajše od teh dveh možnih poti.

Nihče pa nam ne zagotavlja, da so tako izračunane poti res najkrajše poti do mejnih točk. Najkrajša pot do F ne vodi nujno po neposredni povezavi iz točke G; morda se do F bolj splača iti iz druge mejne točke D, morda pa celo prek mejne točke D in še točke E, o razdalji do katere še ne vemo prav ničesar.

Če razmislimo, pa bomo odkrili, da vsaj za eno mejno točko že poznamo tudi najkrajšo razdaljo do nje. Med mejnimi točkami C, D, F in ostalimi, ki na skici niso označene, poiščemo tisto, do katere vodi najkrajša pot iz ene od obiskanih točk. Recimo, da je to točka C: recimo, da je pot od A do C, ki vodi iz H, krajša od katerekoli druge poti do katerekoli druge mejne točke – krajša, torej, od poti prek H ali J do D, krajša od poti prek G do F, krajša od vseh drugih poti do mejnih točk.

Če je tako je pot do C, ki vodi iz H, tudi najkrajša možna pot do C. Res, ne more biti drugače. Če obstaja kakšna še krajša pot, bi morala voditi prek kake druge mejne točke, recimo D. To pa ne more biti, saj že je pot do D daljša kot pot do C-ja – rekli smo, da je C

"najbližja" mejna točka. Do točke C je seveda mogoče priti tudi iz točk, ki niso mejne, a takšne poti bi bile še daljše, saj moramo tudi do teh točk nekako priti, pridemo pa lahko le prek mejnih točk, ki pa so, spet, lahko le daljše.

Zdaj, ko poznamo najkrajšo pot do C, jo lahko *obiščemo*. Rezultat obiska kaže desni del slike. C smo dodali med obiskane točke in si zapomnili, iz katere obiskane točke smo prišli vanj. Vse točke, s katerimi je C povezana, so postale mejne: zanje zdaj poznamo najkrajšo pot do njih, ki vodi prek C (seveda pa to, kot zdaj vemo, ni nujno tudi najkrajša pot do njih). Za obstoječe mejne točke pa se je najkrajša znana razdalja do njih morda skrajšala: najkrajša znana pot iz obiskanih točk do D morda po novem ne vodi več prek H ali J, temveč prek C.

Korak ponavljamo: spet poiščemo najbližjo mejno točko in jo dodamo med obiskane. Postopek lahko končamo, ko obiščemo B, saj nam je s tem znana najkrajša razdalja do nje. Morebitne preostale mejne točke in točke onstran nas ne zanimajo več.

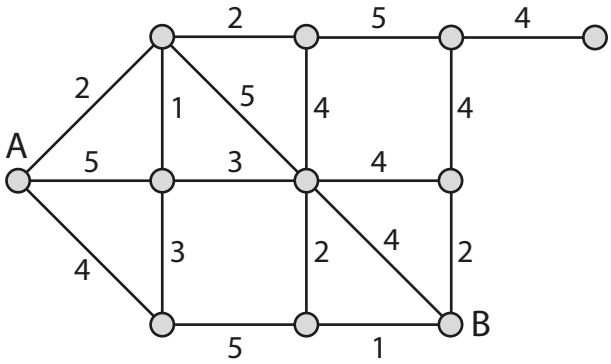
Še enkrat povejmo, ker je pomembno: postopek se ustavi, ko obiščemo ciljno točko in ne že takrat, ko postane mejna. Za mejne točke namreč še ne vemo, ali poznamo najkrajšo pot do njih ali ne. Za obiskane točke pa je najkrajša pot znana.

Če bi ravno hoteli, pa bi lahko postopek gnali še naprej, dokler ne obiščemo vseh točk v grafu. S tem bi dobili *drevo* najkrajših poti iz točke A do vseh drugih točk v grafu (zakaj drevo, bo jasno iz konkretnega primera spodaj). O drevesu smo maloprej povedali nekaj lepega: v drevesu je do vsake točke mogoče priti le na en način. Koren drevesa najkrajših poti je začetna točka, A, torej nam bo drevo povedalo, kako iz A najhitreje priti do vseh drugih točk.

Le še, kako se postopek začne, smo pozabili povedati. A četudi se ne bi spomnili, je menda očitno: začnemo tako, da imamo le eno obiskano točko, namreč začetno točko, A. Pot do nje je dolga 0.

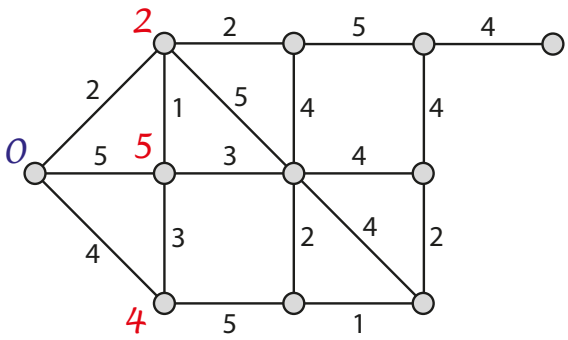
Kaj pa, če ciljno vozlišče iz začetnega sploh ni dosegljivo? Algoritem bo to pravzaprav opazil: zgodilo se mu bo, da bo mejnih točk zmanjkalo, ciljna pa še vedno ne bo obiskana. V tem primeru pač iščemo najkrajšo pot, ki je ni in neuspeh je neizbežen.

In zdaj konkretni primer: poiskati želimo najkrajšo pot od A do B na grafu, ki ga kaže slika.

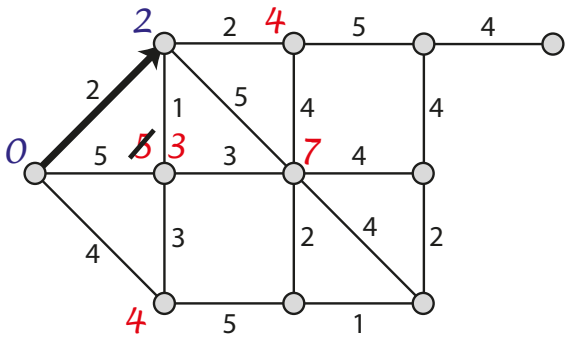


Potem algoritma je ilustriran spodaj.

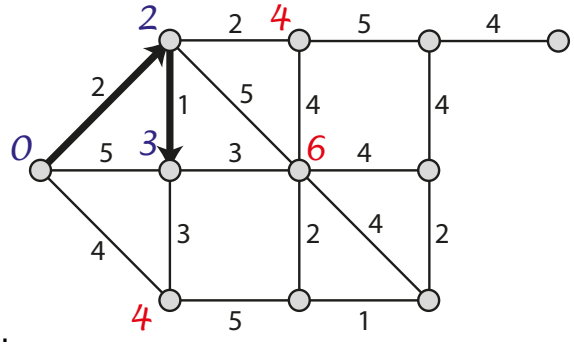
1.



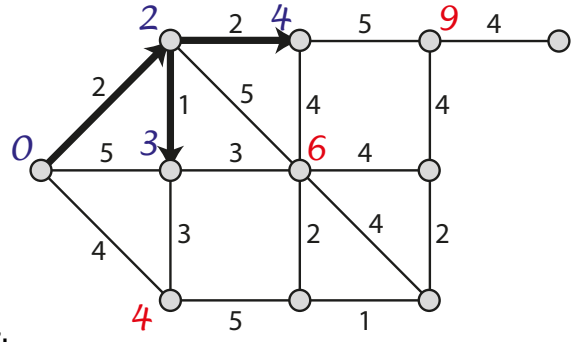
2.



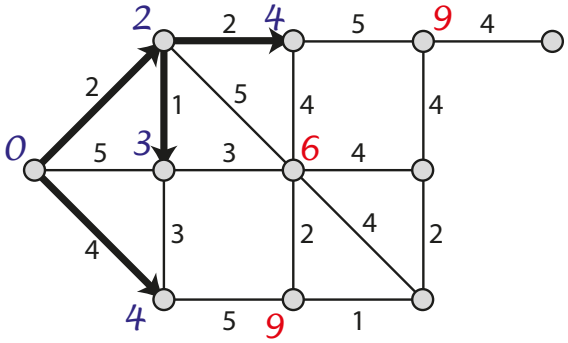
3.



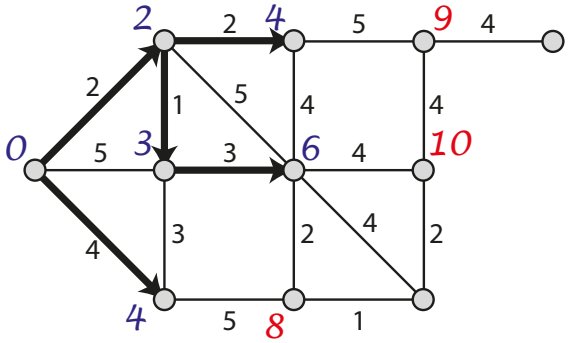
4.



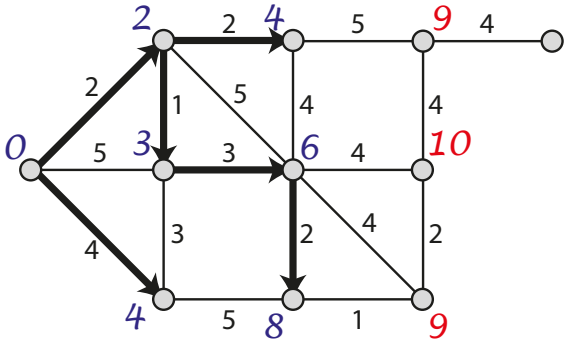
5.



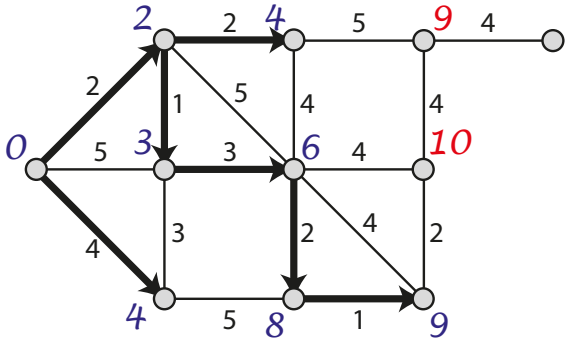
6.



7.



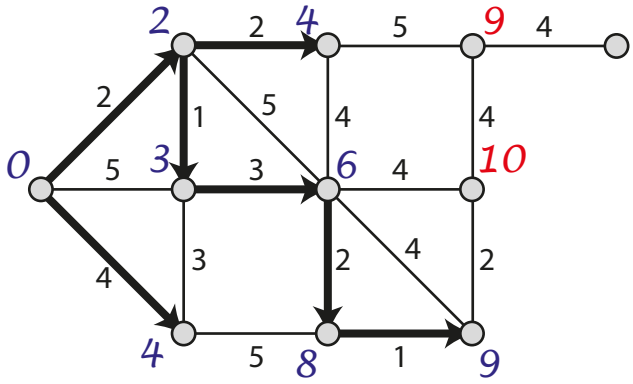
8.



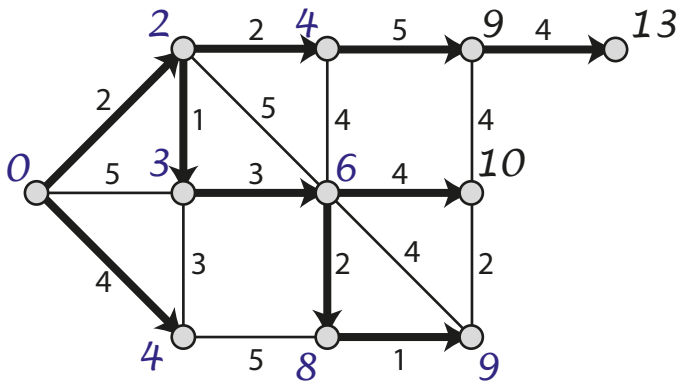
1. V začetku je obiskana točka A, razdalja do nje je 0. Razdalje do treh mejnih točk so 2, 5 in 4.
2. V naslednjem koraku razglasimo najbližjo mejno točko za obiskano. S tem dobimo dve novi mejni točki; razdalji do njiju sta 4 in 7. Do mejne točke, ki je bila prej oddaljena 5, zdaj poznamo krajšo pot dolžine 3.
3. Med obiskane prestavimo mejno točko, oddaljeno 3. To ne prinese novih mejnih točk, le pri eni od obstoječih popravimo razdaljo s 7 na 6.

4. Zdaj imamo dve mejni točki na razdalji 4. Odločimo se za katerokoli od njiju; izbira lahko vpliva na to, kakšna bo najkrajša pot, ki jo bomo našli, ne pa tudi na to, kako dolga bo. Če, recimo, izberemo zgornjo točko, to prinese novo mejno točko na razdalji 9.
5. Med tremi mejnimi točkamim (razdalje do njih so 4, 6 in 9) izberemo najbližjo in jo prestavimo med obiskane. Dobil smo novo mejno točko, razdalja do nje je 9.
6. Zdaj prestavimo med obiskane mejno točko, ki je na razdalji 6. To nam da eno novo mejno točko (na razdalji 10) in zmanjša razdaljo do ene od obstoječih mejnih točk z 9 na 8.
7. Zdaj obiščemo mejno točko, ki je na razdalji 8. To prestavi ciljno točko med mejne točke ... nismo pa še prepričani, da že poznamo najkrajšo razdaljo do nje.
8. Pravzaprav jo: mejne točke so oddaljene 9, 10 in 9. Ciljna točka je najbližja mejna točka (no, ena od njih), torej jo obiščemo. Delo je končano, najkrajša pot in njena dolžina sta znani.

Mimogrede smo izračunali še najkrajše poti do vseh drugih obiskanih vozlišč. Najkrajših poti do ostalih mejnih in do morebitnih vozlišč nismo izračunali in nas tudi ne zanimajo. Lahko pa bi nadaljevali, dokler ne obiščemo vseh vozlišč; tako bi izvedeli najkrajše poti od A do vseh drugih vozlišč.



Če izpustimo povezave, ki nas ne zanimajo, saj ne nastopajo v najkrajših poteh, dobimo drevo.



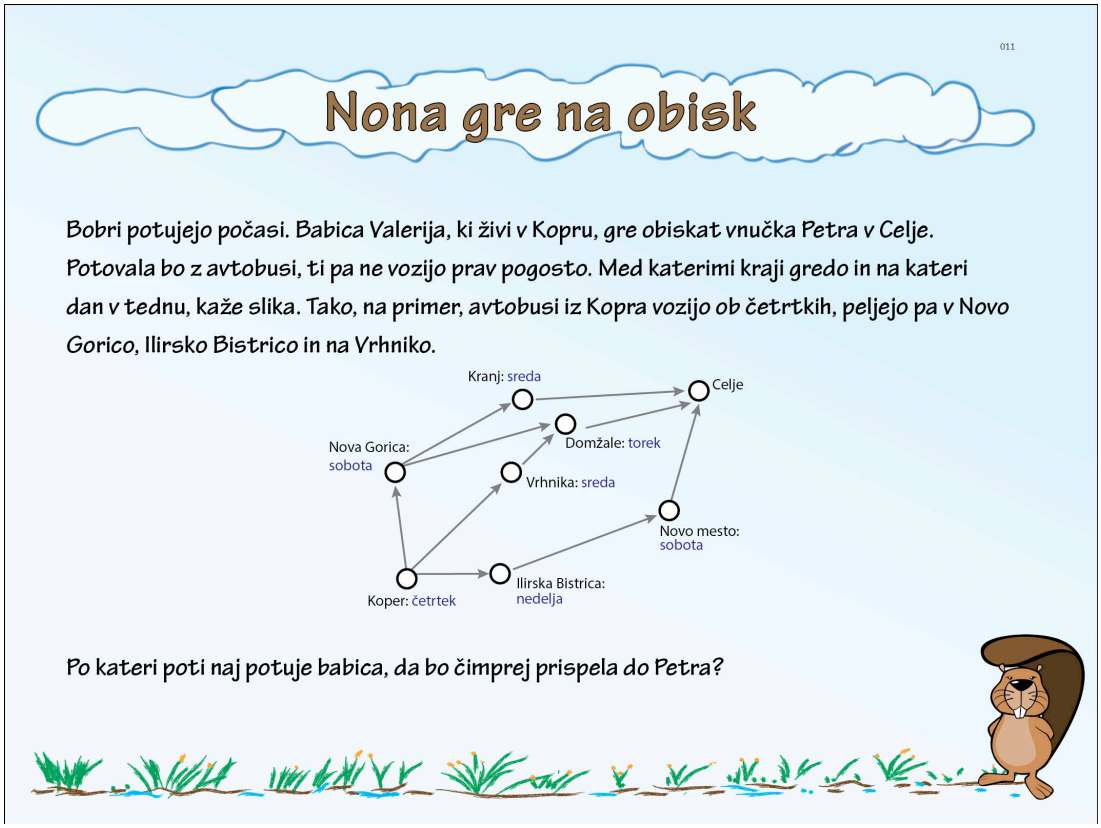
Da mora biti rezultat drevo, je očitno: povezave vodijo do vseh vozlišč (vsaj do vseh vozlišč v tistem delu grafa, ki je dosegljiv iz začetne točke) in do vsakega vozlišča vodi le ena povezava, samo smo vsako vozlišče obiskali (to je, premaknili iz množice mejnih v množico obiskanih točk) le enkrat.

Ne spreglejmo, da drevo govori le o najkrajših poteh iz A v B, ne pa tudi o najkrajših potem med drugimi pari vozlišč. No, med nekaterimi že: najkrajša pot iz vozlišča označenega z 2 do vozlišča z oznako 13 gre gotovo točno tam, kjer jo kaže drevo; če bi obstajala kaka krajša, bi jo uporabili tudi tu. Tudi pot od 2 do 10 vodi tako, kot kaže tole drevo. Pot od 8 do 13 pa vodi bogvekje. Če bi jo hoteli poiskati, bi morali pognati Dijkstrin algoritem iz točke 8.

Dijsktrin algoritem je hiter: čas izvajanja je sorazmeren $M \log N$, kjer je M število povezav in N število točk v grafu. Če predpostavimo, da imajo vse točke bolj ali manj podobno stopnjo, na primer k, bo čas izvajanja sorazmeren $k\,N \log N$: za dvakrat večji graf bo algoritem potreboval nekaj več kot dvakrat daljši čas. O tem se sicer ni težko prepričati, vendar zahteva, da vemo, recimo, kaj je kopica, tega pa ne vedo ne otroci ne mnogi izmed bralcev tega besedila (razen v agronomskem pomenu besede, seveda).

Mimogrede se spomnimo čebele in dinamičnega programiranja. Dijkstrin algoritem je lep primer algoritma, sestavljenega po načelu dinamičnega programiranja. Tako kot smo pri čebeli počasi širili fronto od začetnega cveta proti spodnji vrstici in za vsak cvet opazovali, odkod se nam najbolj splača priti vanj, tudi tu počasi širimo fronto od začetnega vozlišča. Razlika je pravzaprav le v tem, da pri čebeli iščemo maksimum, tu pa minimum, in da imamo pri čebeli dobičke na vozliščih grafa, tu pa ceno na njegovih povezavah. Sicer pa gre za eno in isto reč.

Tule je še lep primer naloge, v kateri je potrebno razmišljati malenkost drugače.



Minimalno vpeto drevo

Včasih nas ne zanima le najkrajša pot iz ene točke v neko drugo ali v vse druge, temveč ... najkrajše poti prek grafa nasploh. Točneje, radi bi zmanjšali število povezav v grafu; graf mora ostati še vedno povezan, skupna dolžina povezav pa čim krajša.

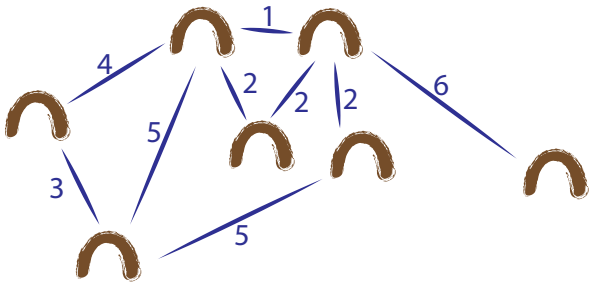
Dijkstrin algoritem nam tu ne pomaga. Potrebujemo nekaj, čemur po svetu pravijo Primov algoritem, Čehi pa vedo povedati, da je to Jarnikov algoritem, ki mu le tujci pomotoma in po krivici pravijo Primov. Pripravljeni pa so, Čehi namreč, na kompromis; pravijo, da bi se temu lahko reklo algoritem DJP, pri čemer je J Jarnik, P pa Prim. Pa D? No, D je Dijkstra, ki se je iste reči spomnil še nekoliko kasneje od prvih dveh. K zmedi pomaga še, da algoritem zamenjujemo z nekim drugim algoritmom za isti namen, s Kruskalovim algoritmom. Da bi bilo vse še bolj zapleteno, je nesrečni Kruskal v istem članku objavil dva algoritma. Le enega od njiju imenujemo Kruskalov, a bi bilo skoraj vseeno, če bi oba, saj sta oba Kruskalova in spet delata isto, le na drugačen način.

Za tiste, ki so ob branju odstavka neuspešno poskušali šteti algoritme, povejmo, da so trije. In jih potolažimo, da smo z imeni končali saj niso pomembna. Sami algoritmi pa so zelo preprosti.

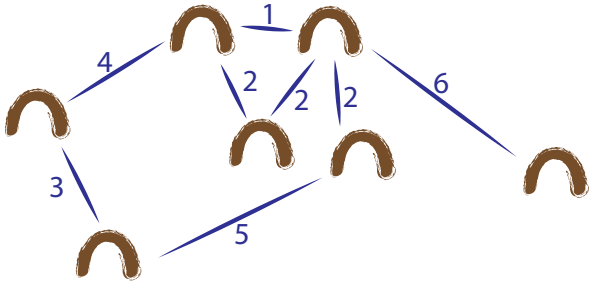
Gredo v eno ali drugo smer. Najprej pokažimo, kako reč teče nazaj. V vsakem koraku pogledamo najdražjo povezavo in se je znebimo, če se je smemo. Povezave se lahko znebimo, če omrežje ostane povezano; če bi ga prekinili, jo pustimo in se lotimo naslednje. Kadar je več enako dragih povezav, se jih lotimo v poljubnem vrstnem redu. Ko ne moremo odstraniti nobene več, nehamo. Izkazalo se bo, da nam vedno ostane ena povezava manj, kot je točk. Kar dobimo, bo vedno drevo, pravimo pa mu *minimalno*

vpeto drevo. *Minimalno* zato, ker ima minimalno ceno, *vpeto* pa zato, ker je vpeto v nek graf.

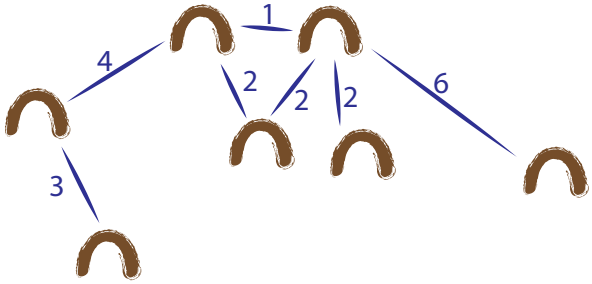
Začnemo torej z grafom takšnim, kot je.



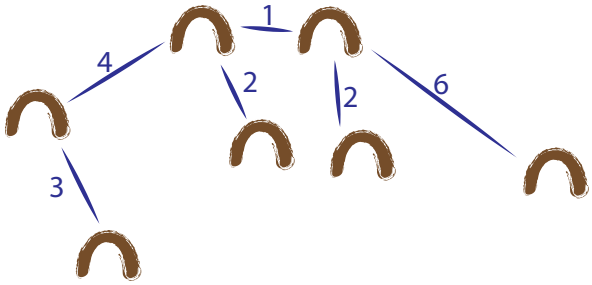
Najdražja povezava ima ceno 6 in z veseljem bi jo pobrisali ... a ne smemo, saj bi s tem odrezali brlog na desni. Nato pogledamo naslednjo najcenejšo povezavo. Dve imamo. Lahko odrežemo levo – bo omrežje razpadlo? Ne bo. Proč z njo!



Pa druga povezava s ceno 5? Tudi brez nje bodo vsi brlogi še vedno povezani, torej se je lahko znebimo.



Lahko odstranimo povezavo s ceno 4? Bognedaj, leva brloga bi ostala brez povezave z ostalimi. Prav, naj ostane. Pa povezava s ceno 3? Tudi ta je nujno potrebna. Zdaj pa povezave s ceno 2. Tri so. Desno potrebujemo, eno od ostalih dveh pa lahko pobrišemo.



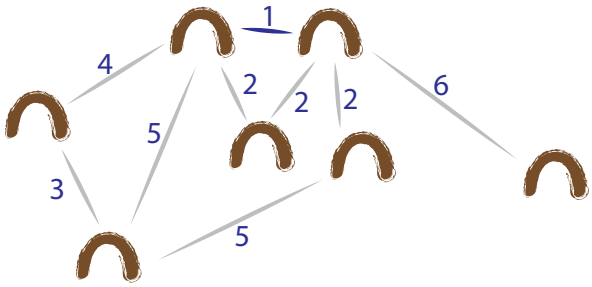
Druge povezave s ceno dve ne smemo pobrisati. Naslednja najdražja povezava je povezava s ceno 1 in ta mora ostati.

Tako smo prišli do konca. Potrebovali bomo 18 metrov žice. Če parafraziramo Tita: z manj ne bo šlo, več ne potrebujemo.

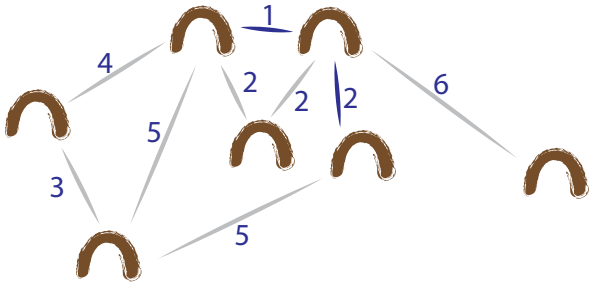
Drugi algoritem deluje v nasprotno smer: namesto da bi začel z vsemi povezavami in jih brisal, jih postopoma dodaja. V vsakem koraku dodamo najcenejšo neuporabljano povezavo, vendar le, če je potrebna. Če bi povezala dve točki, ki sta že povezani (posredno, prek drugih povezav), gremo na naslednjo najcenejšo.

Začnemo z grafom brez povezav; povezave, ki bi lahko bile, a jih še nismo dodali, bomo označili s sivo.

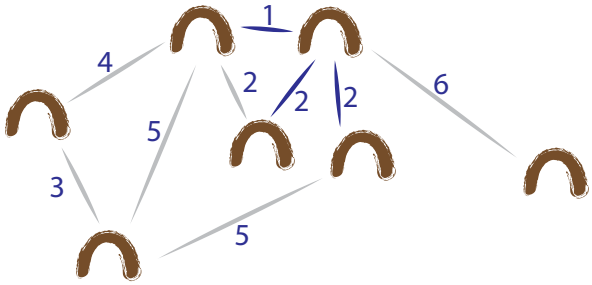
Poiščemo najcenejšo povezavo in jo dodamo. Tule je to za povezava s ceno 1.



Nato dodamo eno od povezav s ceno 2. Katero, je vseeno.

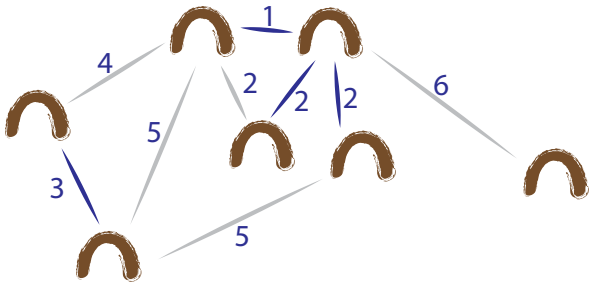


Nato dodamo naslednjo povezavo s ceno 2. Spet je vseeno, katero.

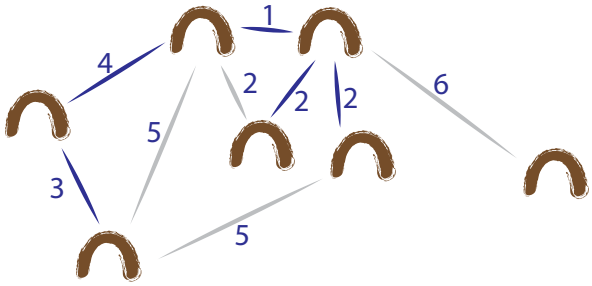


Še eno povezavo s ceno 2 imamo. To pa pustimo, saj bi povezala dva brloga, ki sta povezana tudi brez nje.

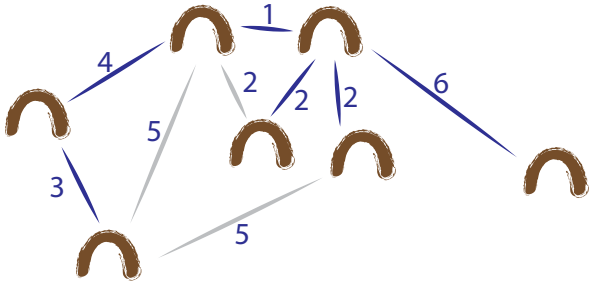
Naslednja najcenejša povezava ima ceno 3. Dodajmo jo.



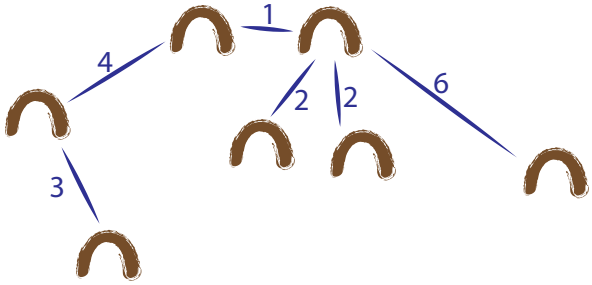
Zdaj je na vrsti povezava s ceno 4. Tudi ta je koristna, saj poveže dva brloga z ostalimi tremi, ki so že povezani.



Naslednja najcenejša povezava ima ceno 5. Pravzaprav sta dve takšni, a nobena nas ne zanima, saj nam ne prineseta ničesar novega: vse, kar bi povezali, je že povezano. Naslednja najcenejša (in edina, pravzaprav že kar najdražja povezava) ima ceno 6. To pa potrebujemo in jo dodamo, cena gor ali dol.



Pobrišimo neuporabljene povezave, da bomo boljše videli, kaj smo pridelali.

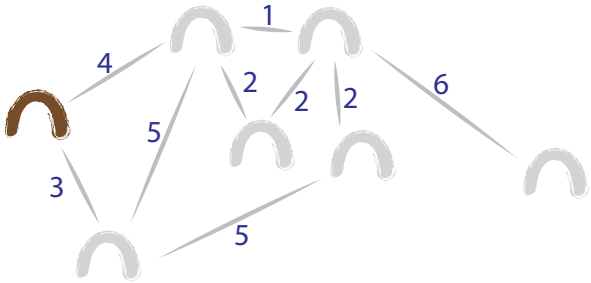


Rezultat je podoben kot prej. Algoritem bo vedno sestavil najcenejše drevo, vendar ne nujno vedno istega. Do razlike pride, ker lahko včasih izbiramo med več enako dragimi povezavami. Končna dolžina povezav pa je enaka, 18.

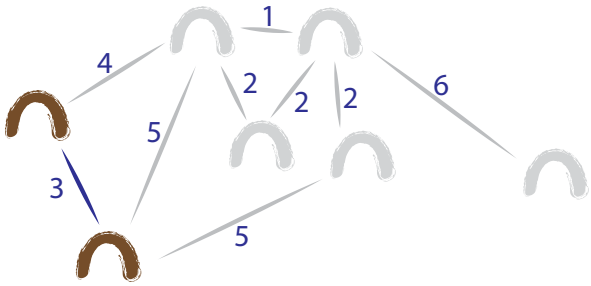
Omenili smo tri algoritme. Tole sta dva. Kje je tretji?

Tudi tretji dodaja, vendar ne povezav, temveč točke (skupaj s povezavami).

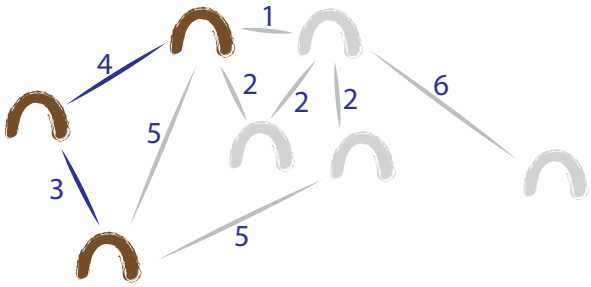
Najprej si izberemo katerikoli brlog. Recimo onega na levi.



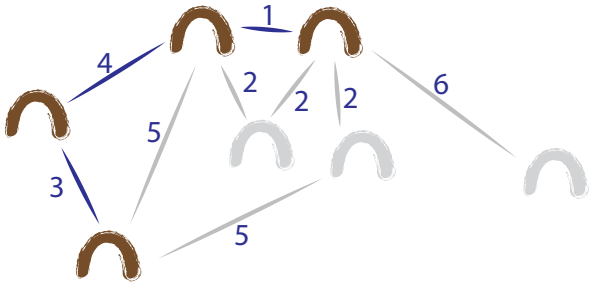
Nato poglejmo, s katerim brlogom ga lahko čim ceneje povežemo. To je brlog pod njim, cena povezave je 3.



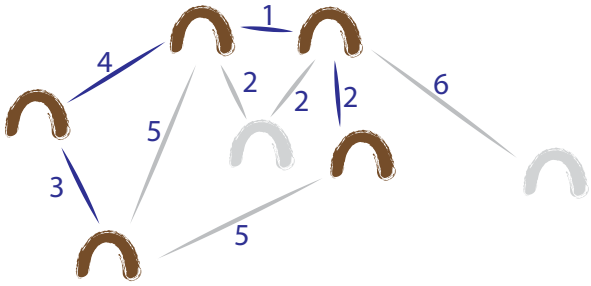
S katerim brlogom lahko najceneje povežemo tadva brloga? Brloga iz izbranih dveh brlogov vodijo tri povezave, njihove cene so 4, 5 in 5. Izberemo tistega s ceno 4.



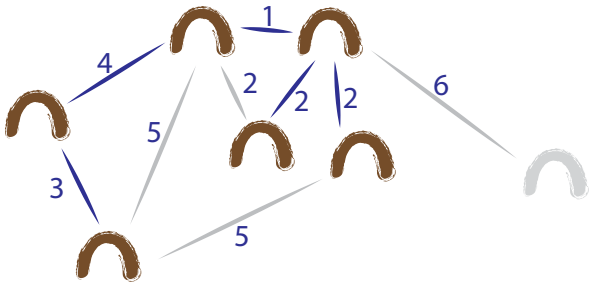
S katerim brlogom bi povezali te tri? Iz njih vodijo tri povezave do brlogov, ki še niso izbrani, njihove cene so 1, 2, 5 in 5. Dodali bomo brlog, do katerega vodi povezava s ceno 1.



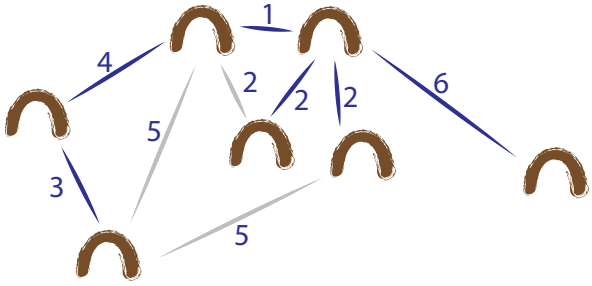
Pa zdaj? Izbrane imamo štiri brloge, do neizbranih treh vodijo povezave s cenami 2, 2, 5 in 6. Izbrali bomo enega od dveh brlogov, do katerega vodi povezava s ceno 2.



Le še dva brloga sta ostala. Do enega pridemo po povezavah z dolžino 2, do drugega po povezavah z dolžino 6. Bolj nam je všeč prvi. (Tule se za hip ustavimo. Zakaj prvi? Navsezadnje bomo morali prej ko slej dodati tudi drugega? Res je, vendar algoritem v tem trenutku še ne ve – ker računalnik pač ne "vidi", kakor vidimo ljudje – ali bo do zadnjega brloga res potrebno uporabiti povezavo s ceno 6, ali pa mora obstaja še kaka cenejša povezava med brlogom, ki ga pravkar dodajamo in zadnjim brlogom.)



Pobožne želje niso bile uslišane in zadnji brlog bomo morali dodati prek povezave s ceno 6.



Končni rezultat je spet tak kot prej. Zadnja dva algoritma sta si v resnici zelo podobna, razlika je le v tem, da pri zadnjem vedno dodajamo v že povezani del, v prejšnjem pa

imamo lahko nekaj časa nepovezane dele, ki se šele v naslednjih korakih povežejo med seboj.

Problem iskanja najmanjših vpetih dreves je zanimiv, ker obstajajo zanj trije različni algoritmi in vsi so preprosti, vsi dajo vedno optimalen rezultat (čeprav ne nujno enakega). Za otroke je pri večini nalog verjetno najpreprostejši postopek z brisanjem. Postopek z dodajanjem pa je prikladen, kadar graf ni podan vnaprej: včasih naletimo na nalogo, ko so podane le koordinate točk, dovoljene pa so vse povezave, med poljubnim parom točk. Tedaj bomo najprej dodali najbližji par točk, nato točk, ki jima je najbližja in tako naprej – tako, kot smo videli v zadnjem algoritmu.

Iskanju minimalnih vpetih dreves je posvečena tudi ena od aktivnosti na Vidri: http://vidra.fri.uni-lj.si/pomagajmo-cestarjem.

Obhodi

Poleg najkrajših poti po grafu nas pogosto zanimajo tudi kake druge, recimo najdaljše. Vsi smo že kot otroci reševali naloge vrste "Prehodi vse poti v parku", tako da boš na vsako stopil le enkrat, ali pa obišči vsa drevesa v parku, a mimo vsakega smeš le enkrat. Zdaj vemo, da lahko te parke narišemo kot grafe in po vsem, kar smo se tule naučili doslej, bi rekli, da morajo najbrž obstajati tudi nekakšni postopki, s katerimi te naloge sistematično rešujemo?

Pot, ki gre prek vsake stezice natančno enkrat, imenujemo Eulerjeva pot ali, če se konča tam, kjer se je začela, Eulerjev obhod. Algoritem zanj je preprost in ga otroci odkrijejo sami. Za začetek recimo, da v vsako križišče vodi sodo število poti. V tem primeru

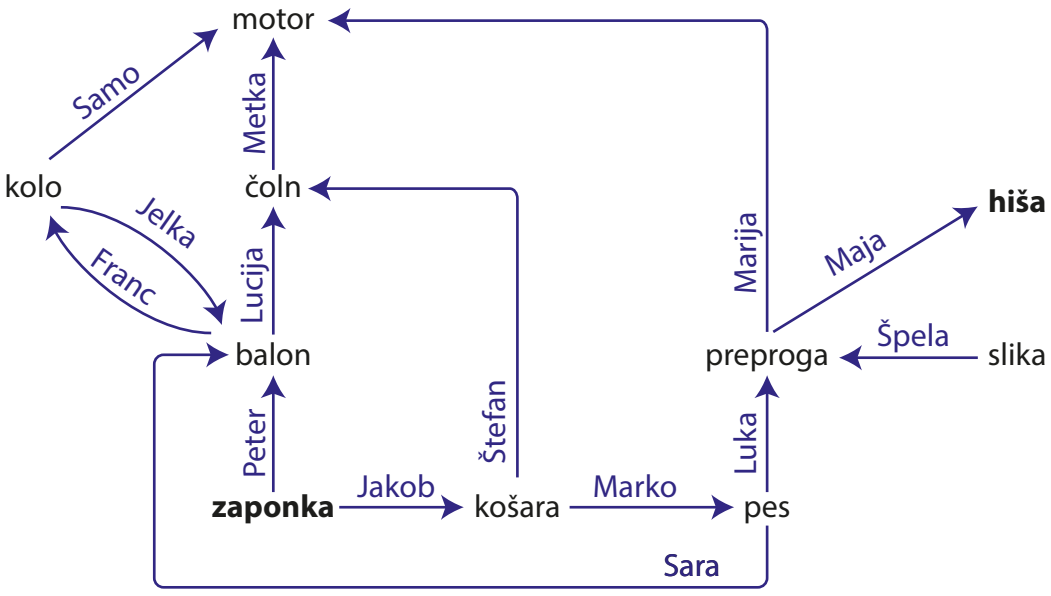
nalogo rešimo tako, da začnemo v kateremkoli in hodimo naokrog, dokler moremo in kakor hočemo. Če se bo kje ustavilo in ne bomo mogli naprej, se bo gotovo nekoč, ko se bomo vrnili v začetno vozlišče. Drugače ne more biti: vsa vozlišča imajo sodo število poti in če smo prišli v neko vozlišče, smo gotovo uporabili liho število poti, ki vodijo vanj oz. iz njega, torej mora gotovo obstajati vsaj še en izhod. Izjema je začetno vozlišče: ko pridemo vanj, je uporabljeno sodo število poti okrog njega. Če se torej zataknemo v njem, obdržimo pot, ki smo jo naredili, a jo popravimo tako, da vanjo "vstavimo" kako od poti, ki je še neizkoriščena.

Če imamo v grafu (ali parku) tudi križišča z lihim številom poti, potem smo lahko prepričani, da takšno križišče ni le eno, temveč vsaj dve. (V resnici jih je vedno sodo število, a to tule ni pomembno.) Pot moramo, kot vedo že otroci, začeti v enem od vozlišč z lihim številom poti, končali pa ga bomo v drugem. Če je križišč z lihim številom poti več, recimo štiri, je naloga nerešljiva.

Drugi problem, obiskati vsa križišča, se imenuje po Hamiltonu – Hamiltonova pot oziroma Hamiltonov obhod. Tu nismo posebej pametni: no, smo, matematiki so si domislili kup izrekov in računalnikarji kup algoritmov, v resnici pa ne poznamo algoritma, ki bi učinkovito poiskal Hamiltonovo pot ali obhod. Čim je graf nekoliko večji, bo algoritem potreboval ogromno časa, da bo poiskal pot. Ne le, da takšnega algoritma ne poznamo, temveč imamo kar dobre razloge, da verjamemo, da ga ni.

Namesto, da bi tule razpravljali o algoritmih, raje poglejmo dva lepa primera nalog, ki ju je lahko rešiti. Na Bobru se občasno pojavljajo naloge, v katerih iščemo Eulerjevo ali Hamiltonovo pot, vendar so precej dolgočasne: navadno je graf že narisan (le da mu v nalogi ne rečemo graf, temveč zemljevid ali kaj podobnega) in je potrebno le poiskati želeni tip poti. Nalogi, ki ju bomo videli tule, sprašujeta po drugačni poti; lepi sta, ker graf ni očiten na prvi pogled.

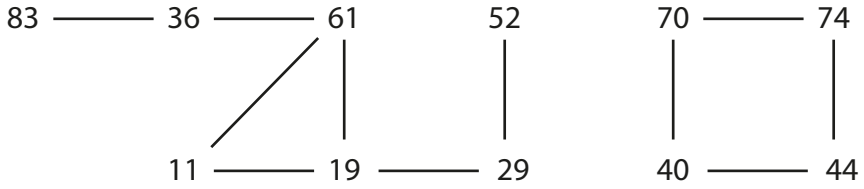
Prva je naloga s trgovanjem. Nalogo je težko reševati zato, ker se iz tabele ne znajdemo. Prerišemo jo lahko v spodnji graf.



Čim pridemo do sem, je reč trivialna: zaponko je potrebno zamenjati za košaro, to za psa, psa za preprogo in preprogo za hišo.

Lepota naloge je v tem, da mora učenec poiskati primerno predstavitev problema in potem je vse enostavno. V kontekstu Bobra je sicer nekoliko nerodna, ker je morda vseeno hitrejše poiskati rešitev s poskušanjem po tabeli kot izgubljati čas s prerisovanjem. A nič hudega: v takšnih primerih lahko izven tekmovanja vidijo, kako se takšno reč reši lepo, sistematično.

Lepota naloge je v tem, da graf ni podan eksplicitno (kot recimo pri Telefonski mreži), niti niso ekplicitno podane relacije (kot pri prejšnji nalogi). Na prvi pogled tole niti od daleč ne diši po grafih. Po drugi strani: kako naj se človek loti te reči? Ne gre drugače, kot da si narišemo hiše in povežemo tiste, med katerimi sme iti poštar, se pravi te, ki imajo kako skupno številko.



V jeziku tega predavanja je to spet, očitno, graf in to, kar iščemo, je najdaljša Hamiltonova pot po grafu. Graf ni povezan (če poštar začne, recimo, pri 29, ta dan ne more iti do 40), zato prave Hamiltonove poti ne bomo našli. Najboljše, kar lahko sestavimo, je 83 – 36 – 61 – 11 – 19 – 29 – 52 ali obratno.

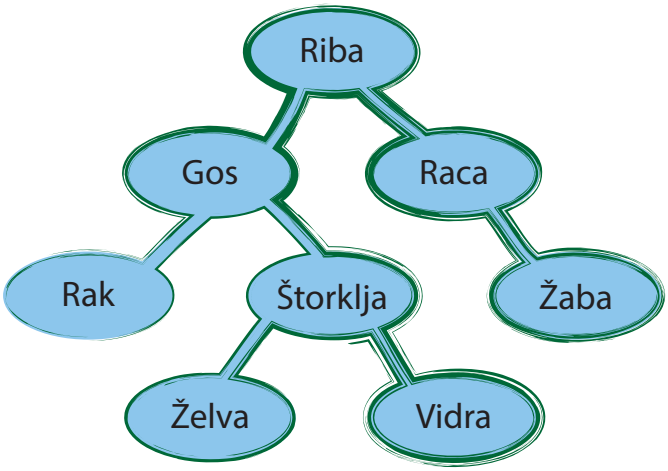
Tako kot prejšnja postane tudi ta naloga trivialna, če se spomnimo na grafe in jo primerno narišemo.

Obhodi dreves

Nekoliko drugačna vrsta obhodov gre prek nekoliko drugačne vrste grafov – dreves, ki smo jih spoznali v prejšnjem sklopu. Čeprav ne sodi čisto k algoritmom, jo omenimo tu.

Ker je tema pomembna, se naloge te vrste pogosto pojavljajo na Bobru. Po drugi strani jo je težko osmisliti izven precej globjega konteksta, za katerega tu ni časa, pa tudi pri Bobru ne pride do izraza – naloge te vrste so pogosto puste, saj nimajo prave zgodbe, temveč govorijo le o bobrih, ki po kakšnih čudnih in z ničemer utemeljenih vrstnih redih obiskujejo svoje prijatelje, raziskujejo jame in podobno.

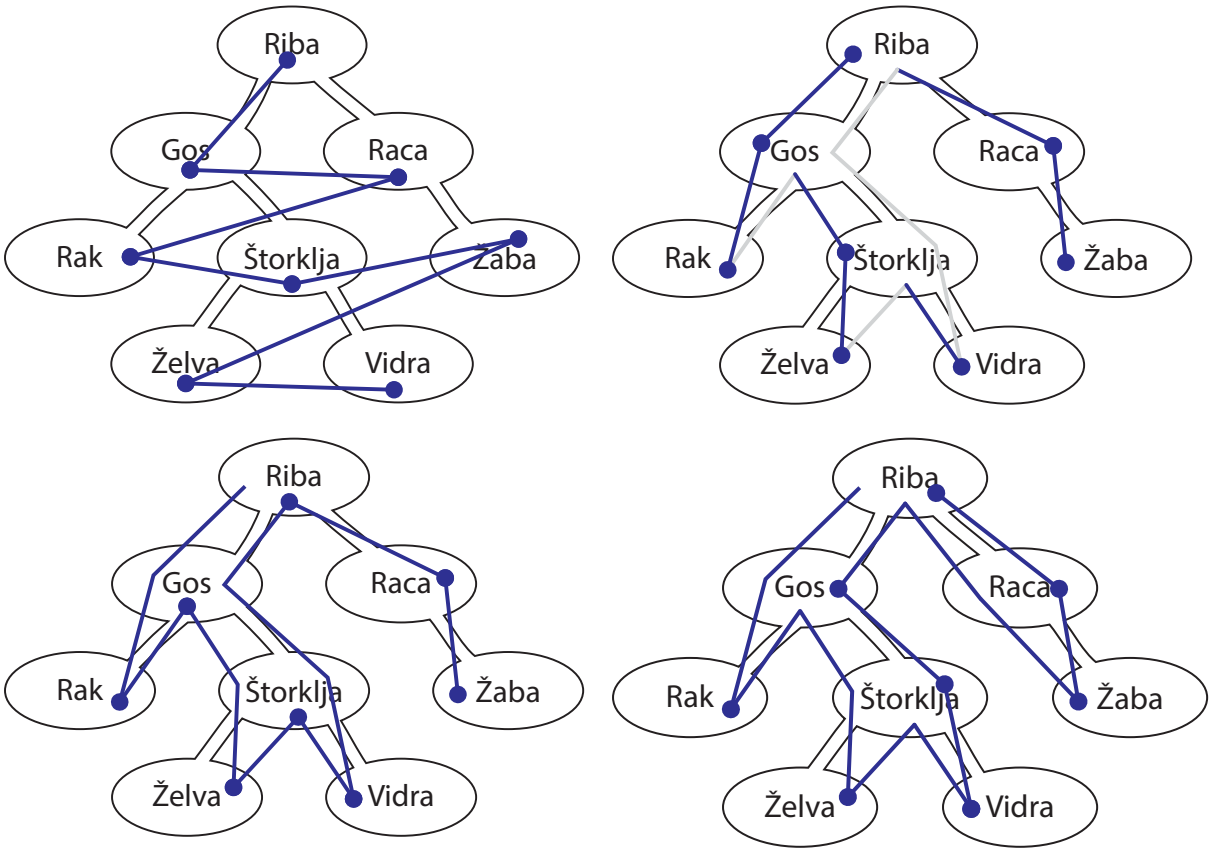
Sicer pa reč ni zapletena. V osnovi gre za tole: recimo, da v mlakah, povezanih s kanali, živijo različne živali, tako kot kaže spodnja slika.



Kako bi našteli prebivalce mlak? Sistematično, po vrsti – po kakršnikoli vrsti, ki se vam zdi smiselna glede na drevo (torej, kriterij za vrstni red naj narekuje drevo, ne abeceda ali kaj podobnega)?

Večina jih najbrž našteje takole: riba, gos, raca, rak, štorklja, žaba, želva, vidra. Nekateri bi šli po drugem sistemu: riba, gos, rak, štorklja, želva, vidra, raca, žaba. Lahko pa tudi tako: rak, gos, želva, štorklja, vidra, riba, raca, žaba. Ali: rak, želva, vidra, štorklja, gos, žaba, raca, riba.

Aha, pravite, da je prvi vrstni red je očiten in edini normalen, naslednji pa vedno bolj čudni ali celo čisto nerazumljivi? No, poglejmo. Za začetek jih narišimo – pri zadnjih dveh je brez slike res težko videti, za kakšen vrstni red (če sploh kakšen?) gre.



Prvi vrstni red imenujemo preiskovanje v širino: najprej "preiščemo" prvi nivo, nato drugega, tretjega in tako naprej, dokler ne pridemo do konca. Že, da govorimo o "preiskovanju" ne naštevanju ali čem podobnem, nakazuje, da prihaja reč iz nekega drugega vica, vendar zanj nimamo časa.

Pri ostalih treh gremo najprej v globino. Po drevesu (ali mlakah) se v vseh treh primerih sprehajamo v enakem vrstnem redu: bober se vedno najprej zapelje v levo, nato v desno mlako, potem pa vrne nivo višje. Razlika je v tem, kdaj "imenujemo" žival v posamezni mlaki.

Predstavljamo si bobra-akviziterja, ki se s čolnom vozi po kanalih med mlakami (dasiravno so bobri odlični plavalci, na tekmovanju Bober skoraj vedno uporabljajo čolne) in jim poskuša prodati kakije ali pa polivinilaste vrečke, kakor je navada pri akviziterjih. (V originalni nalogi je zgodba nekoliko drugačna; tole je različica za odrasle, ki črpa iz sloga socialnega realizma.)

Prvi vrstni red je popolnoma neuporaben, saj mu je čoln le v napoto – namesto da bi se z njim vozil med kanali, ga bo moral skupaj s kakiji in vrečkami vlačiti po kopnem med mlakami. Tudi za programiranje (recimo, da bi imeli takšno drevo shranjeno v pomnilniku in bi morali izpisati živali) je ta vrstni red najbolj zoprn.

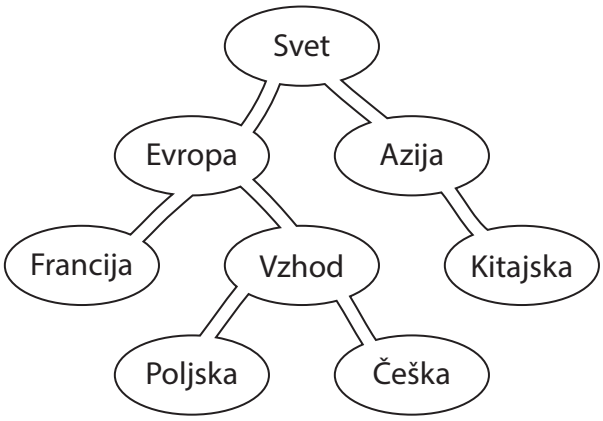
V drugem vrstnem redu (prvem izmed teh, ki gredo v globino, ne širino) se bober vozi med mlakami. Ko pride do določene mlake, najprej nadleguje žival, ki živi v tej mlaki, nato vse živali na levi strani in nato one na desni. Ko se vrača, živali, ki jo je že nadlegoval, ne nadleguje ponovno – tega tudi resnični akviziterji pretežno ne počnejo.

V tretjem vrstnem redu najprej nadleguje vse živali na levi, nato žival, ki živi v mlaki in nato vse na desni. Če gremo od začetka: najprej se bo lotil živali, ki so levo od ribe, nato ribe in potem živali desno od ribe. Na vsaki strani se zgodba ponovi: med živalmi levo od ribe bo najprej nadlegoval tiste, ki so levo od gosi, nato gos in potem vse, ki so desno od gosi. Levo od gosi je le rak, desno od gosi pa spet ponovi isto: najprej živali levo od štorklje (torej želvo), nato štorkljo in nato živali desno od štorklje (vidro).

V zadnjem vrstnem redu najprej nadleguje živali levo od posamezne mlake, nato živali desno in končno še žival v mlaki. Tako bo, recimo, najprej nadlegoval tiste levo od ribe, nato tiste desno in na koncu še ribo samo. Najlepše se vrstni red vidi spodaj, kjer se loti najprej želve, nato vidre in šele na koncu štorklje.

Tudi ti trije vrstni redi imajo imena: prvi je *premi*, drugi *vmesni* in tretji *obratni*. Lažje razumljive tujke jim pravijo *prefiksni*, *infiksni* in *postfiksni*: zapomnimo si jih po tem, da povedo, kje se pojavi "koren", žival, ki jo srečamo prvo – pred ostalimi (*pre*-), med njimi (*in*-) ali za njimi (*post*-). Nobene škode pa ni, če si imen – ne latinskih ne kranjskih – sploh ne zapomnimo, saj gre vendar samo za imena.

Namesto tega vse čudne vrstne rede – vse razen prvega, ki menda ni čuden – upravičimo, ovsakdanjimo. Tule je razdelitev držav po svetu.

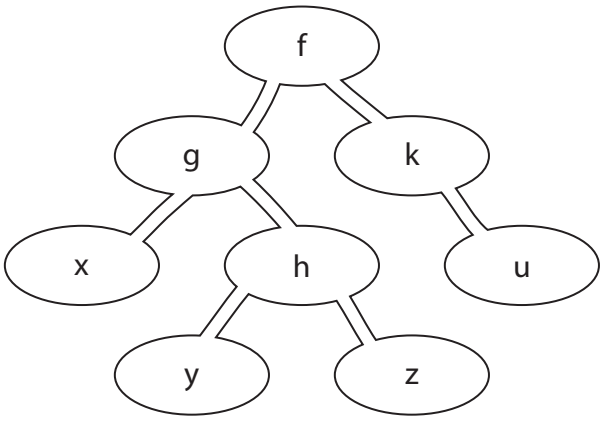


Če bi jo hoteli prepisati v (gnezdene) alineje, bi rekli

- Svet
    - Evropa
        - Francija
        - Vzhod
            - Poljska
            - Češka
    - Azija
        - Kitajska

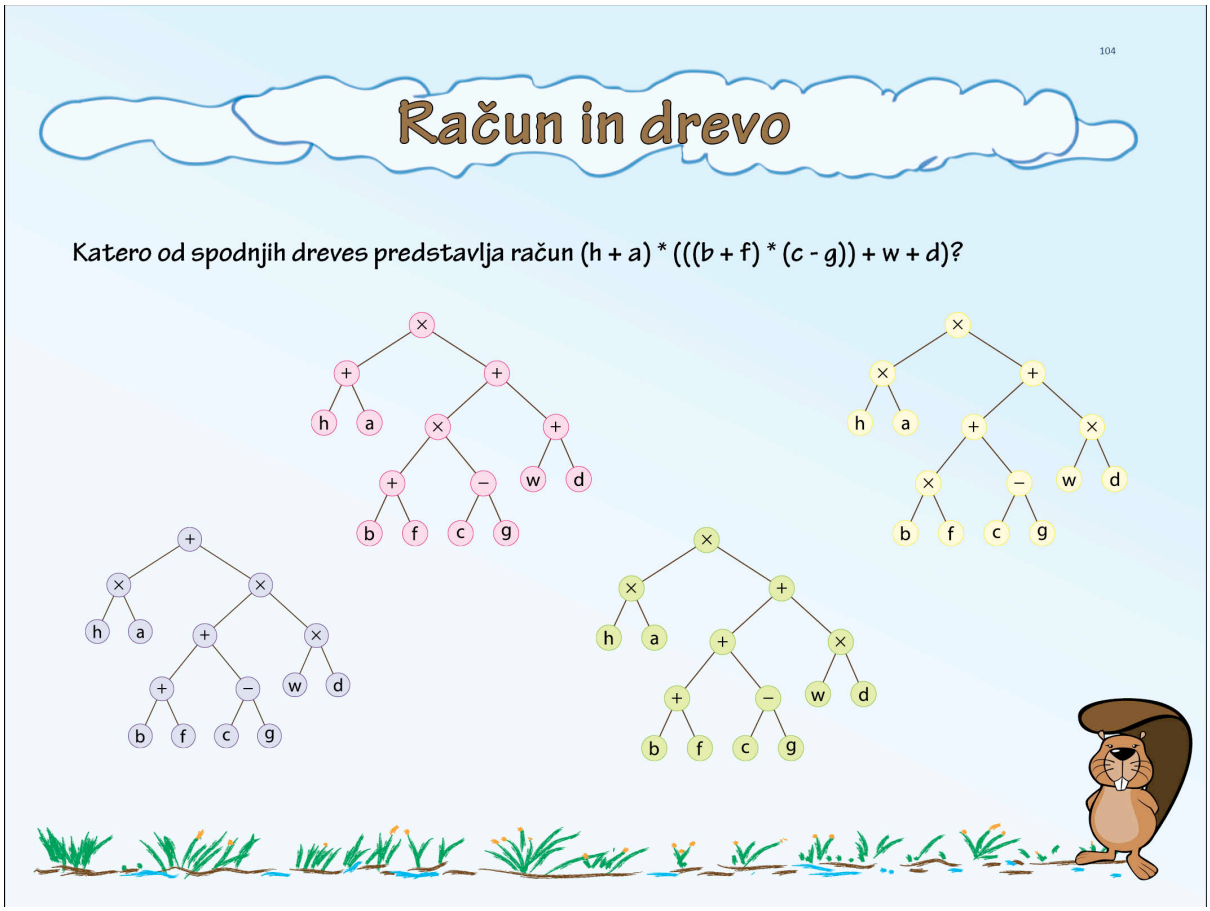
Vrstni red, v katerem smo izpisali drevo, je natančno takšen kot v drugem razporedu.

Tu je še malo drugačen primer, v spomin na nalogo z aritmetičnimi izrazi. Recimo, da imamo funkcije *f*, *g*, *h* in *k* ter spremenljivke *x*, *y*, *z* in *u*. Tedaj bi drevo



ustrezalo izrazu $f(g(x, (h(y, z)), k(u))$. Vrstni red, v katerem se pojavljajo simboli, f, g, x, h, y, z, k, u je natančno drugi vrstni red.

Za utemeljitev tretjega načina se spomnimo naloge



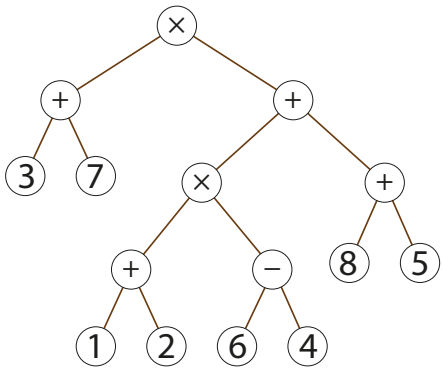
(Da se ne mučimo ponovno: pravilno je rožnato drevo.) V izrazu (h + a in tako naprej) se najprej pojavi levo poddrevo, nato simbol, nato desno poddrevo. Predstavitev izrazov z drevesi pravzaprav niti ni tako slaba ideja: izraz (h + a) * (((b + f) * (c – g) + w + d) je jako nepregleden. Kar dobro ga moramo pogledati, da odkrijemo, katera je najbolj zunanja operacija. Iz drevesa pa je povsem očitno, da bomo nekaj zmnožili – namreč tisto na levi s tistim na desni. Na levi je vsota h in a, na desni pa vsota ... uf, tistega, kar je na levi in tistega, kar je na desni te vsote.

Za zadnji vrstni red najprej recimo, da akviziter ve, da bo vsaka žival kupila toliko kakijev, kot oba njena soseda skupaj. Da bi lahko določeni živali (recimo gosi) povedal, koliko sta kupila njena soseda (rak in štorklja), bo moral najprej k raku in štorklji, šele nato k želvi.

Zgodba zveni privlečena za lase, pa ni: kako bi v resnici izračunali vrednost izraza $f(g(x, (h(y, z)), k(u))$ (ker ga je težko brati, morda raje poglejte drevo, ki ga predstavlja)? Lahko najprej izračunamo vrednost funkcije f, nato g in h? Ne. Prav tako ne moremo najprej izračunati g, nato f, nato h. Ne, najprej je potrebno izračunati to, kar je levo (g) in kar je desno (h); nato iz rezultata tega dvojega izračunamo f.

Zadnji vrstni red je od vseh treh zato najbolj naraven – čeprav nas je v začetku morda najbolj zmedel. Se opravičujem, a tako je. To je vrstni red, v katerem računamo.

Razlike med zapisi še enkrat povejmo na primeru izrazov, le da črke zamenjajmo s številkami, da bo nazorneje.

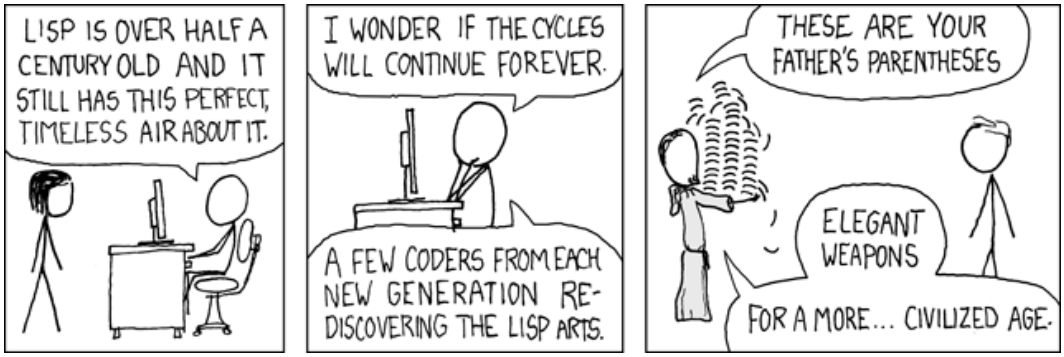


(3 + 7) * (((1 + 2) * (6 – 4) + 8 + 5).

Tega, infiksnega zapisa smo tako vajeni, da se nam zdi, da drugačni zapisi nimajo smisla. Obstajajo pa računalniški jeziki, v katerih bi morali takšen izraz zapisati kot

(* (+ 3 7) (+ (* (+ 1 2) (- 6 4) (+ 8 5)))),

torej najprej operator in nato operandi. V tej obliki je izraz zapisan prefiksno. To se po svoje lepo bere: zmnoži vsoto 3 in 7 ter vsoto zmnožka vsote 1 in 2 in razlike 6 in 4 ter vsote 8 in 5. Edini problem tega jezika in takšnega opisa je, da potrebujemo oklepaje (in to, očitno, veliko oklepajev, predvsempa enako zaklepajev, ki se naberejo na koncu!), da jasno določimo vrstni red. Ni pa povsem nenaravno. (Povejmo: gre za jezik Lisp in njegove naslednike. Lisp je eden najstarejših, a tudi najvplivnejših in, za razliko od njegovega opešanega vrstnika Fortrana, še vedno atraktivnih jezikov. Prednost takšnega zapisa je, da vodi v preproste jezike (po neki definiciji preprostosti, ki je jasna predvsem ljubiteljem takšnih jezikov;

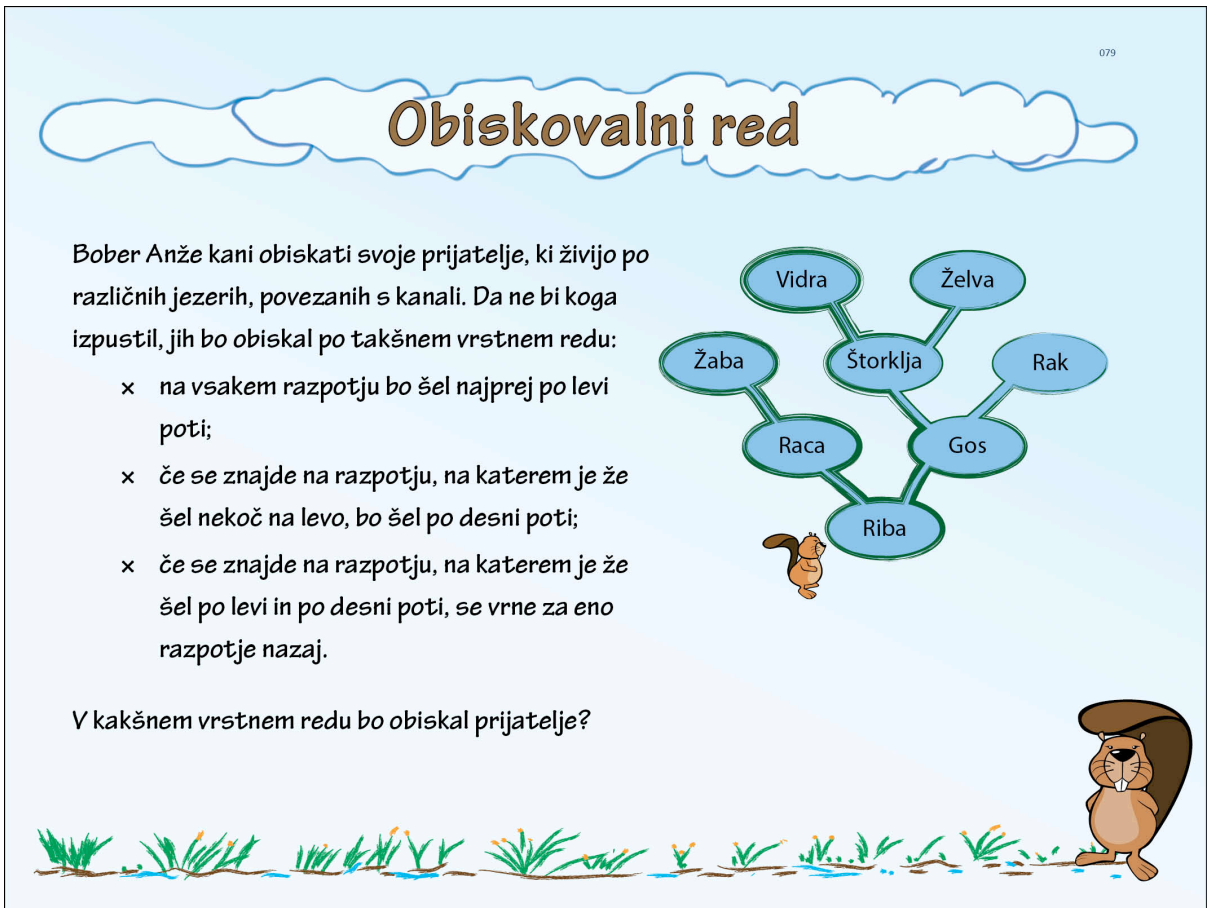


(Vir: http://xkcd.com/297/; strip je pod licenco CC-BY-NC, dovoljena uporaba v nekomercialne namene)

).

Obstaja pa še tretji način zapisa izrazov: 3 7 + 1 2 + 6 4 - * 8 5 + + *. Ta zapis ustreza zadnjemu postfiksnemu. V tem vrstnem redu računamo v resnici: vzamemo 3 in 7 ter ju seštejemo ter si zapomnimo vsoto. Vzamemo 1 in 2 ter ju seštejemo. Vzamemo 6 in 4 ter ju odštejemo. Zmnožimo zadnji dve številki... Lepota tega zapisa je v tem, da zanj nikoli ne potrebujemo oklepajev.

Ste med kupovanjem tiskalnika ali nameščanjem gonilnikov zanj kdaj naleteli na besedo PostScript? PostScript je jezik, v katerem računalnik pove tiskalniku, kaj in kako naj izpiše. Ime je dobil po postfiksnem zapisu vsega – ne le aritmetičnih izrazov, ves jezik je narejen tako kot nemščina, z glagoli na koncu. Tudi datoteka PDF ni nič drugega kot program, ki v jeziku PostScript pove bralniku (na primer Acrobat Readerju), kaj naj nariše in kakšno besedilo pokaže.



V tej nalogi, na kateri je temeljil razdelek, je drevo obrnjeno drugače. Tako je praktično, saj sta leva in desna s tem natančno določeni: če rišemo drevo od zgoraj navzdol, kot je običaj med računalnikarji, bober potuje proti gledalcu in njegova leva je naša desna, kar zmede opis.

Kot smo omenili na začetku razdelka, je zoprna lastnost teh nalog, da nimajo posebne zgodbe: čudaški bober se je odločil za tak in tak vrstni red obiskovanja mlak. Zakaj, naloga ne pove.

Druga nerodna reč ob teh nalogah je, da od otrok zahtevajo bolj ali manj samo, da razumejo opis postopka. Če ga razumejo, je rešitev trivialna; če ne ... pa je otrokovih težav morda kriv samo nejasen opis. In nekatere vrste obhodov je v resnici težko lepo

opisati. Pisec tega besedila čuti do Bloomove taksonomije in podobnih reči zmeren odpor, tule pa jo morda omenimo – če ne zaradi drugega, vsaj zato, da učitelji ne bodo imeli občutka, da so se o tem piflali zaman (dasiravno bi bil ta občutek morda na mestu): tovrstne naloge zahtevajo predvsem razumevanje in, v najboljšem primeru, čisto preprosto uporabo. Od tekmovalnih nalog bi si človek upal pričakovati več.

Žal pa so precej pogoste, zato ne bo nič narobe, če jih otrokom pokažemo. Da jim bomo imeli ob tem povedati še kaj zanimivega, pa smo v tem razdelku kar obilno poskrbeli.

Barvanje grafov

Naloga je navidez podobna nalogam o Pavlinih ploščicah in o socialnih omrežjih, ki smo jih reševali, ko smo spoznavali grafe. Saj tudi je. Ob grafih jih omenjamo zaradi algoritma, povezanega s tovrstnimi grafi.

Francis Guthrie je sredi 19. stoletja barval zemljevid angleških grofij in odkril, da ga že s štirimi barvami lahko pobarva tako, da sta sosednji grofiji vedno različnih barv (pri tem grofiji, ki se stikata le v eni točki nista sosednji; poleg tega nobena grofija ni sestavljena iz več kosov, kot recimo Hrvaška, ki jo bosanski Neum preseka v dva dela; nadalje moramo predpostaviti, da je zemljevid planaren, torej ni na krogli ali čem še bolj čudnem, recimo torusu). Opazil je, da to ni kaka posebnost angleških grofij, temveč to velja za vsak zemljevid – vsaj tako se mu je zdelo, vendar tega ni mogel dokazati. Guthriejev brat je to pokazal znanemu matematiku de Morganu, ta pa je to omenil v pismu, ki ga je leta 1852 napisal Hamiltonu, ki smo ga že srečali na naši poti v prejšnjem razdelku.

Čemu ta zgodba? Ker je zanimiva: problem je matematike frustriral dobrih 120 let, preden so leta 1976 končno dokazali, da je s štirimi barvami res mogoče pobarvati vsak (ravninski) zemljevid. Ker so si pri dokazovanju morali pomagati z računalnikom, so matematiki nad dokazom še dolgo negodovali.

Na Bobru so pogoste naloge, ki zahtevajo barvanje zemljevidov. Da uvidimo zvezo med zemljevidi in grafi, smo zdaj menda že dovolj pametni. Kako pridemo iz zemljevida do grafa, je povedala naloga: vsaka pokrajina je točka in dve točki sta povezani, če imata skupno mejo. S tem seveda še nismo rešili problema, le prevedli smo ga na grafe. Zdaj moramo vsaki točki prirediti barvo, a tako, da sta povezani točki vedno različnih barv.

Algoritem? Za ročno reševanje (in tudi reševanje z računalnikom) se splača začeti pri točki, ki je najbolj zapletena, ki ima največ povezav. Dodelimo ji neko barvo, nato pa – po občutku – nadaljujemo z drugo najbolj zapleteno točko ali pa s kako točko, ki je povezana s to in ima tudi sama veliko povezav.

Drugo, na kar se splača paziti so polni podgrafi. Če v grafu zalotimo štiri točke, ki so povezane med sabo, vemo, da bodo morale biti štirih različnih barv. Barvanje lahko začnemo s takšnimi štirimi točkami in potem nadaljujemo po poti, ki nas najbolj "omejuje", se pravi tako, da vedno pobarvamo tisto točko, pri kateri imamo najmanj izbire.

Tale opis, kot prvo, ni ravno opis algoritma, saj računalnik ne more delati "po občutku". Poleg tega ta algoritem ne da nujno optimalne rešitve. Lahko se zgodi, recimo, da bomo porabili pet barv. A nič hudega: če najdemo rešitev s petimi barvami, vemo (hvala, matematiki!), da se nismo dovolj potrudili. Rešitev popravljamo, dokler ne najdemo takšne s štirimi barvami.

Lepo, vendar: kakšno zvezo ima to z računalništvom? In kakšen pomen ima, pravzaprav, to na splošno? Je vredno izgubljati čas z barvanjem zemljevidov?

Barvanje zemljevidov je le začetek zgodbe. Na zemljevide lahko pozabimo: predstavljajmo si, da imamo kar nek graf, sestavljen kakorsižebodi in česarsižebodi, naša naloga pa ga je pobarvati tako, da so vse povezane točke različnih barv in pri tem uporabiti čim manj različnih barv. V splošnem bomo namreč potrebovali več kot štiri, to je jasno: le predstavljajte si poln graf z osmimi točkami – ker je vsaka povezana z vsako, bomo potrebovali nič manj kot osem barv. Ker v tem primeru *ne vemo* vnaprej, da bomo potrebovali le štiri (ali celo le tri ali dve) barvi, je naša naloga bistveno težja. Imamo ogromen graf in pobarvamo ga s sedmimi barvami; kako lahko vemo, da ga ni mogoče tudi s šestimi?

Če je kdo pričakoval, da mu bomo, tako kot pri najkrajših poteh in najmanjših vpetih drevesih spet postregli z algoritmom, se moti: tudi za ta problem ne poznamo učinkovitega algoritma in verjamemo, da ga tudi nikoli ne bomo, ker ga ni.

Tečnež iz zadnje vrste pa se spet oglasi: kakšno zvezo ima to z računalništvom ali čemerkoli drugim? Čemu izgubljati čas z barvanjem grafov?

Barvanje grafov je morda še pomembnejši algoritem od iskanja poti, iskanja najkrajših poti, vpetih dreves... Na problem barvanja grafov lahko prevedemo kup zelo različnih problemov. Da ne bomo predolgi, vas tule le usmerim na gradivo: na strani http://vidra.fri.uni-lj.si/ubogi-geograf, ki sicer opisuje aktivnosti, ki jih lahko na to temo izvajamo z otroki, boste mimogrede izvedeli tudi, kako lahko z barvanjem grafov sestavljamo urnike in kako tudi reševanje sudokuja ni nič drugega kot eno samo duhamorno barvanje grafov.

Pokritje

Zadnji problem s področja grafov bomo le še omenili – toliko, da vemo zanj in da ga ne zamenjujemo s kakim minimalnimi vpetimi drevesi.

V grafih pogosto iščemo različne oblike pokritij. Med vsemi vozlišči želimo, recimo, poiskati takšno podmnožico vozlišč, da se vsaka povezava dotika enega izmed označenih vozlišč. Recimo, da želimo razpostaviti policiste po križiščih tako, da bo stal policist vsaj na enem koncu vsake ulice (lahko pa tudi na obeh). Ob tem pa želimo, da je policistov čim manj; takšnemu pokritju rečemo minimalno pokritje.

V obrnjeni različici naloge postavljamo policiste na ulice in želimo, da v vsako križišče vodi vsaj ena ulica, ki ima policista. Spet v tretji različici postavljamo policiste na križišča in želimo, da za vsako križišče velja, da ima bodisi svojega policista bodisi je policist v enem sosednjih križišč.

Za naloge iskanja minimalnih pokritij – zdaj bi lahko že uganili, da je tako, ne? – nimamo dobrih algoritmov. Tudi za te probleme velja, da s številom točk v grafu čas reševanja zelo hitro narašča in da verjamemo, da hitrejši algoritmi za ta problem ne obstajajo.

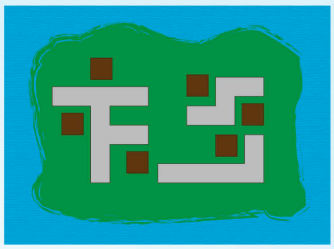
Naloge na Bobru zato lahko rešujemo le po zdravi pameti, z opazovanjem. Naloge s pokritji so – tako kot gornja – žal pogosto sestavljene tako, da je že na prvi pogled jasno, da gre za graf, tako da učenec pri reševanju ne potrebuje posebnega prebliska, temveč le bistre oči in koncentracijo.

Eden od problemov pokritij je predstavljen tudi na Vidri, v aktivnosti povezani s Piranskimi sladoledarji.

Problemov pokritij si lahko izmislimo, kolikor hočemo. Tole je različica, ki ima poleg vsega še cene. Rešujemo jih lahko, vsaj na Bobru, le po občutku.

Cestne svetilke

Bobri bi radi razsvetlili Bobrograd. Na zemljevidu na desni so s temno barvo označene hiše, ulice so sive; ostale povezave so podzemne. Bobri bi radi razsvetlili (nadzemne) ulice. Na voljo so svetilke, ki svetijo eno, dve ali tri smeri. Njihove cene so različne:




5 bevrov


6 bevrov


7 bevrov

Koliko najmanj bo stala osvetlitev?

Borut Batagelj

Programi kot zaporedja ukazov

Uvodna aktivnost

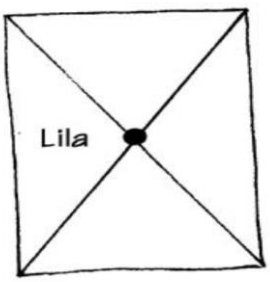
Računalniki vedno naredijo to, kar jim naročimo. Če smo pri dajanju navodil – se pravi programiranju – nepazljivi, so lahko rezultati napačni. Kako se počutita programer in računalnik bomo spoznali v naslednji aktivnosti, ko bomo prevzeli vlogo programerja s tem, da bomo dajali navodila za risanje, ter vlogo računalnika, ko bomo poskušali narisati sliko po navodilih drugega.

S pomočjo te aktivnosti spoznamo kako težko je podajati dobra navodila in kako smešni so lahko rezultati ohlapnih navodil. Na ta način lahko učencem prikažemo, zakaj pišemo programe v posebnih **računalniških jezikih**, ki programerja silijo v točno izražanje.

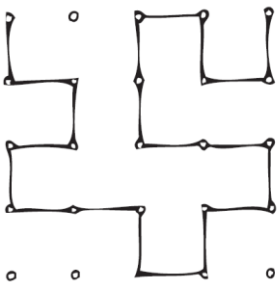
Primer enostavne naloge

Udeleženci naj vzamejo papir in pisalo. Preberem jim spodnja navodila, oni pa sproti rišejo.

1. Narišite majhen krog sredi papirja.
2. Pobarvajte krog.
3. Potegnite črto iz zgornjega levega kota papirja skozi krog v spodnji desni kot.
4. Potegnite črto iz zgornjega desnega kota papirja skozi krog v spodnje levi kot.
5. Napišite svoje ime v trikotnik levo od kroga na sredini papirja.
6. Udeleženci naj pokažejo, kaj so narisali.
7. Na tablo narišem, kar bi morali, glede na zgornja navodila narisati.



Slika 9



Na takšen način lahko učenci prakticirajo še druge slike (slika 9), s tem da en otrok narekuje navodila za risanje, drugi pa rišejo. Več primerov slik ter še druge zanimive aktivnosti s področja računalništva lahko najdete na portalu: http://vidra.si.

Takšno igro vsakodnevno igrajo tudi programerji in računalniki. Programer ima podobno nalogo kot učenec, ki opisuje sliko. Tudi programer daje navodila računalniku in šele potem vidi, kaj je računalnik na osnovi teh navodil naredil. Največji problem v igri niso nenatančna, temveč dvoumna navodila. Zaradi tega so lahko nastale različne slike, ki niso bile takšne, kot bi morale biti.

Ker so človeški jeziki (slovenščina, angleščina, stara grščina ...) preveč ohlapni, nejasni, dvoumni, za programiranje računalnikov uporabljamo posebne jezike, ki so jasnejši in programerja silijo v točno izražanje.

Teoretično ozadje opisane aktivnosti

Računalniku dajemo navodila v obliki **programov**. Vsak program opravlja določeno nalogo. Programi so napisani v jezikih, ki imajo omejen nabor ukazov. Različni programski jeziki so primerni za različne naloge: nekateri so primernejši za programe, ki tečejo na spletu, drugi jeziki so znani po tem, da je mogoče v njih zelo hitro programirati manj zmogljive programe, spet v tretjih je programiranje težje in počasnejše, zato pa so programi, napisani v njih, zelo hitri.

Ne glede na izbrani jezik mora biti programer previden in zelo točno povedati računalniku, kaj bi rad od njega. Računalnik bo vedno dobesedno izpolnil ukaze (kadar bo to mogoče, seveda), pa čeprav je rezultat lahko smešen.

Programerji morajo biti natančni, saj ima lahko že drobna napaka v programu resne posledice. Predstavljajte si, kaj se lahko zgodi zaradi napake v programu, ki krmili jedrsko elektrarno, prižiga luči na železniških semaforjih ali vozi letalo! Napakam v programih rečemo hrošči v čast hrošču (točneje molju), ki so ga našli v enem prvih elektronskih računalnikov iz štiridesetih let prejšnjega stoletja. Odstranjevanju hroščev iz teh ogromnih računalnikov so rekli razhroščevanje (*debugging*) in tudi današnji programerji razhroščujejo svoje programe, pri čemer uporabljajo posebna programska orodja, ki jim pravijo razhroščevalniki.

Programiranje za otroke

Zgornjo aktivnost lahko nadaljujemo tudi tako, da sedaj poskušamo računalniku podati navodila, da izriše želeno sliko na ekran. Seveda je izbira pravega programskega jezika za učence, ki niso vešči programiranja, ključnega pomena. Na srečo imamo veliko izbiro tako imenovanih vizualnih orodij za programiranje, kjer ukaze enostavno skladamo skupaj, kot so otroci tega že navajeni pri Lego kockah.

Eno najbolj poznanih in razširjenih okolij je gotovo programsko okolje Scratch (http://scratch.mit.edu/).

Scratch

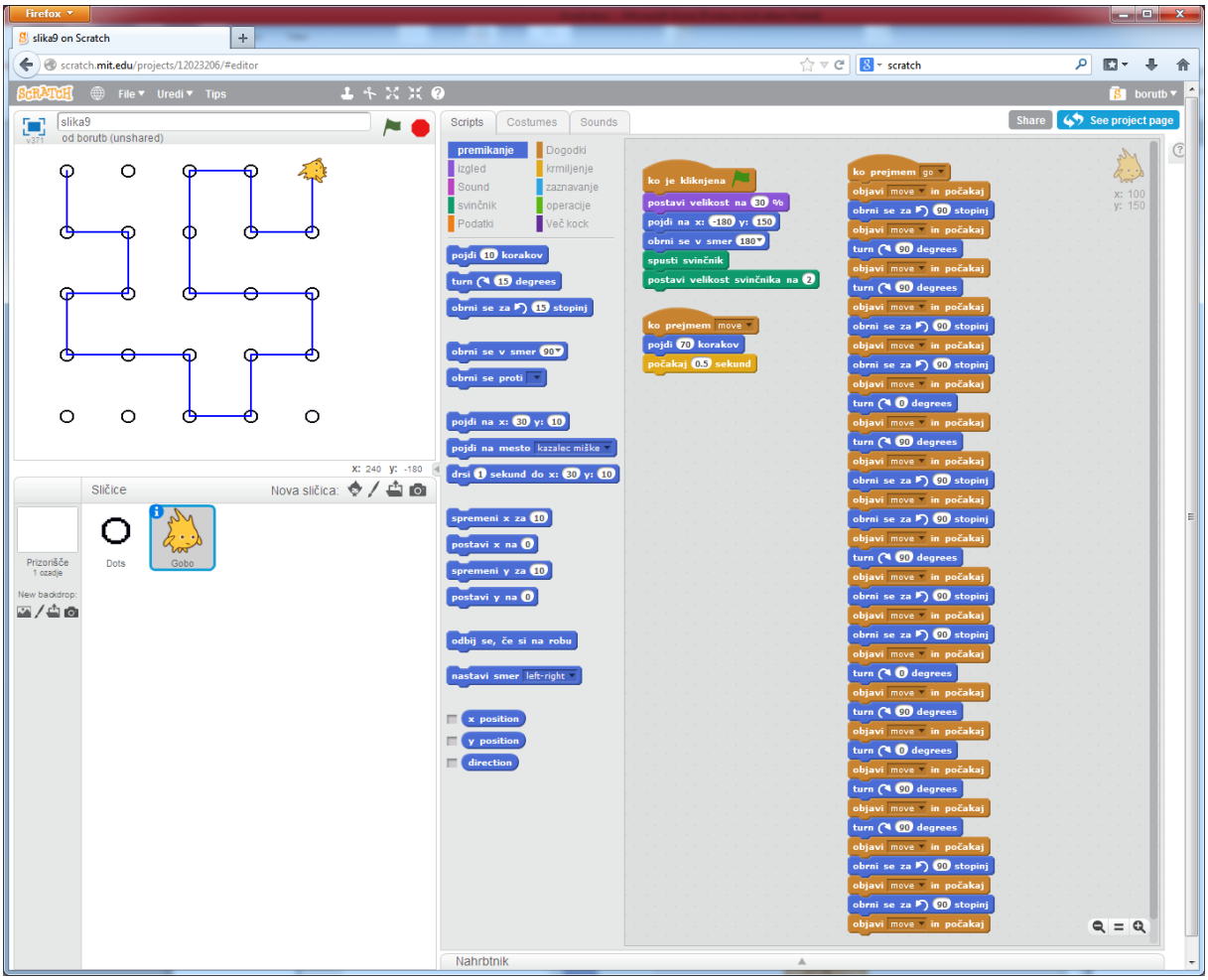
Prednost programskega jezika Scratch je v tem, da je uporabniški vmesnik preveden v slovenščino, čeprav ima prevod še nekaj pomanjkljivosti. Tudi literaturo že lahko najdemo v slovenskem jeziku, na internetu pa imamo ogromno že narejenih programov, ki jih lahko spreminjamo. Kot je značilno za vizualna okolja, v Scratchu ni tipkanja, ni prevajalnika in podobnih strašljivih orodij in pravzaprav ne moremo napisati programa, ki ne bi deloval. Programirate namreč tako, da v program vlečete gradnike, ki program sestavljajo. Gradniki so različnih vrst – od premikalnih (premakni predmet, zavrti se ...) do upravljalnih (ponavljaj n-krat, ponavljaj dokler), sestavljate pa jih lahko le na način, ki zagotavlja pravilno delovanje. Ukaz lahko odložite le na mesto, kjer ima smisel, pri tem pa vam pomaga oblika programskih gradnikov, ki nakazuje, kam določen ukaz spada.

Glavni elementi Scratcha so bitja in predmeti, ki jim v angleščini pravimo *sprite*. Na odru spremljamo premikanje figur, igre in animacije. Položaj figure na odru je določen s pomočjo koordinatnega sistema. Figuro lahko narišeš sam ali pa uporabiš katerekoli slike s svojega računalnika. Figuri lahko napišeš **program**. Figuri lahko zamenjaš videz. Vsaka figura ima lahko tudi svoj seznam **zvokov.** V sklopu multimedije lahko uporabljaš: urejevalnik slik, kamero in snemalnik zvoka. Ukazi so razdeljeni v osem skupin, ki združujejo vsebinsko podobne ukaze. Ukazi znotraj skupine so iste barve.

Koncepti, ki jih Scratch podpira: zaporedje ukazov, zanke, pogojni stavki, spremenljivke, tabele, odzivi na dogodke, vnos podatkov preko tipkovnice, naključna števila, logične operacije. Težje pa boste predstavili podprogram, rekurzijo, dedovanje, definiranja lastnih razredov ter branje in pisanje iz datoteke.

Za izdelavo programov ne potrebujete nameščanja nobenih dodatnih programov, ker razvojno okolje deluje kar v spletnem brskalniku. Tako izdelane programe lahko poganjate v razvojnem okolju ali pa jih zapakirate v spletno aplikacijo.

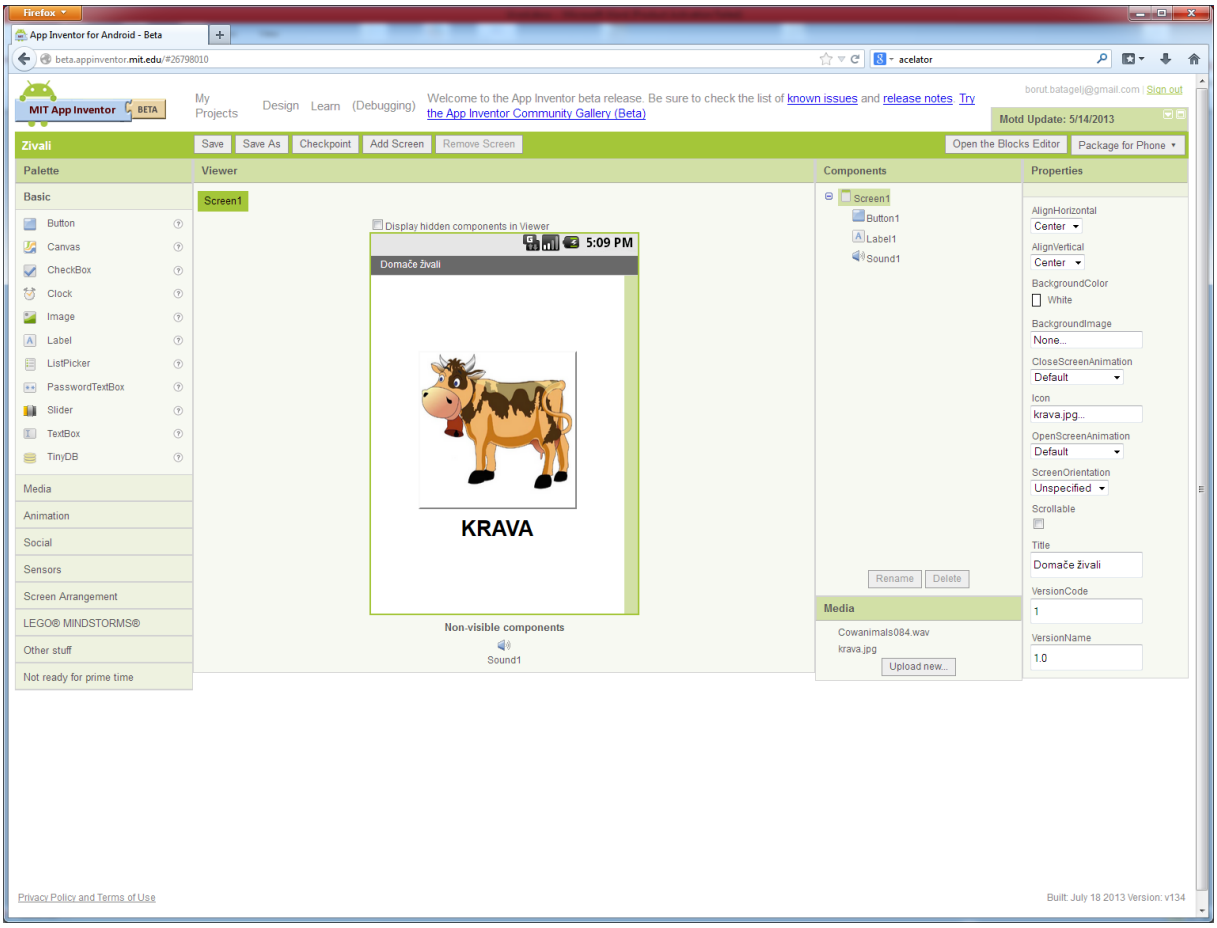
Primer programa v Scratchu, ki zriše sliko 9 zgornje aktivnosti:



App Inventor

Če vas je okolje Scratch navdušilo, potem boste okolje App Inventor oboževali. S pomočjo Scratcha namreč lahko izdelujete programe za računalnik, App Inventor pa vam omogoča, da izdelate čisto pravo igrico za svoj telefon ali tablico, ki uporablja Android. Pri programiranju imate na voljo vse dodatne senzorje, ki jih prenosna naprava omogoča: kamero, mikrofon, ekran na dotik, GPS, senzor premikov. Tako lahko izdelate zelo zanimive interaktivne programe.

Primer programa: zivali.apk, izvorna koda: zivali.zip


Firefox
App Inventor for Android - Beta
beta.appinventor.mit.edu/#26798010
acelator
MIT App Inventor
BETA
My Projects
Design
Learn
(Debugging)
Welcome to the App Inventor beta release. Be sure to check the list of known issues and release notes. Try the App Inventor Community Gallery (Beta)
borut.batagelj@gmail.com | Sign out
Motd Update: 5/14/2013
Zivali
Save
Save As
Checkpoint
Add Screen
Remove Screen
Open the Blocks Editor
Package for Phone
Palette
Basic
Button
Canvas
CheckBox
Clock
Image
Label
ListPicker
PasswordTextBox
Slider
TextBox
TinyDB
Media
Animation
Social
Sensors
Screen Arrangement
LEGO® MINDSTORMS®
Other stuff
Not ready for prime time
Viewer
Screen1
Display hidden components in Viewer
5:09 PM
Domače živali
KRAVA
Non-visible components
Sound1
Components
Screen1
Button1
Label1
Sound1
Rename
Delete
Media
Cowanimals084.wav
krava.jpg
Upload new...
Properties
AlignHorizontal
Center
AlignVertical
Center
BackgroundColor
White
BackgroundImage
None...
CloseScreenAnimation
Default
Icon
krava.jpg...
OpenScreenAnimation
Default
ScreenOrientation
Unspecified
Scrollable
Title
Domače živali
VersionCode
1
VersionName
1.0
Privacy Policy and Terms of Use
Built: July 18 2013 Version: v134

Teorija: programski jeziki

Za prvo programabilno napravo bi lahko šteli Jacquardove mehanske statve iz leta 1801, s katerimi je omogočal tkanje različnih vzorcev z različnimi programi (luknjane kartice). Pravi program in programiranje pa se je začelo z zasnovo računalniškega modela, ki ga je predlagal John von Neumann leta 1964. Pred tem je bilo možno računalnike reprogramirati samo s pomočjo žic, priključkov ali stikal. V spominu so bili shranjeni samo podatki, ne pa ukazi. Za vsak problem je bilo tako potrebno prežičiti celoten računalnik. Prvi elektronski splošno namenski računalnik - ENIAC je imel na primer 6000 stikal, ki jih je bilo potrebno spremeniti za drugo nalogo. Von Neumann je predlagal, da bi bili ukazi, ki kontrolirajo računalnik, zapisani poleg podatkov v spominu. Tako bi bilo za rešitev novega problema potrebno samo preurediti ukaze, namesto razporejati žice ali stikala – to pomeni, napisati nov program. Tako lahko rečemo, da je programiranje, kot ga poznamo dandanes, izumil von Neumann.

V nadaljevanju si bomo pogledali, kakšen sploh je računalnik, kot si ga je zamislil von Neumann in ki se še danes uporablja, ter kaj sploh je to program. Nato se bomo sprehodili skozi zgodovino in poskušali razumeti, kako so višje-nivojski programski jeziki sploh nastali. Zanimalo nas bo, kako jih računalnik pravzaprav razume, če pa so na nivoju našega jezika, računalnik pa kot vemo, pozna samo enice in ničle.

Računalnik

Računalnik je naprava, ki izvaja računanje in procesira podatke. Računalnik dela pod kontrolo **programa** – množice navodil, ki povejo, kaj mora računalnik delati. Strojna oprema opisuje elektroniko in mehanične dele računalnika. Programska oprema pa so programi, ki kontrolirajo strojno opremo.

Računalnik sestavljajo: izhodne naprave (tiskalnik, ekran, zvočniki), vhodne naprave (tipkovnica, miška, mikrofon, skener, kamera), primarni pomnilnik (hrani podatke in programe, podatki se izgubijo, ko ugasnemo računalnik) in sekundarni pomnilnik (trdi disk, CD, tračne enote, USB ključi, SD kartice), centralna procesna enota CPU, ki skrbi za izvajanje ukazov, aritmetično logična enota ALU, skrbi za računanje in primerjanje, preko signalov sporoča tudi drugim napravam.

Poznamo dve vrti programov: aplikativni in sistemski. Aplikativni skrbijo za določeno nalogo. Sistemski pa naredijo računalnik sploh uporaben. Najpomembnejši takšen program je operacijski sistem (OS).

Program

Računalniški program je nabor **navodil**, ki usmerjajo obnašanje računalnika. Programiranje je umetnost in znanost oblikovanja in pisanja programov. Je kreativna in zabavna aktivnost reševanja problemov. Svojo rešitev lahko preizkusite v obliki izvajajočega programa.

Od strojnega jezika do višje-nivojskih jezikov

Danes se večinoma programira v višje-nivojskih jezikih kot so Java, C++ ali Python. Programski jezik označimo kot višje nivojski, če imajo stavki programa pomen tudi v naravnem jeziku. Vsi našteti programski jeziki imajo na primer pogojni stavek IF, ki ga uporabimo kot: IF pogoj THEN akcija. Če je izpolnjen določen pogoj, izvedi sledečo akcijo. Nekateri programski jeziki imajo lastnosti, ki jih naredijo primerne za pisanje specifičnih programov: COBOL za komercialne programe, FORTRAN za inženirje in znanstvenike ter C in C++ za programe operacijskega sistema. Poleg tega pa uporabljajo notacijo in simbole, ki so ljudem razumljivi. Aritmetične operacije so na primer predstavljene z operatorji +, -, * in /, tako da lahko zapišemo v programu izraz: (a+b)/2.

Pojavi pa se težava, ker računalniki takšnega izraza neposredno ne razumejo. Če hočemo, da bo računalnik razumel tako navodilo, moramo program prevesti v računalniku razumljiv **strojni jezik**, ki ga razume CPU (Centralno procesna enota) oziroma mikroprocesor. Vsak procesor ime svoj strojni jezik, zato tudi imamo programe za različne naprave (operacijske sisteme). Takšni programi se imenujejo **platformsko odvisni programi**.

V splošnem strojni jezik bazira na binarni kodi, ki ima dve stanji, to je 0 ali 1. Tako so vsi ukazi in podatki predstavljeni z binarnimi kodami. Seštevanje dveh števil na primer predstavimo z ukazom opcode: ADD (011110), ki prejme 3 operande, ki predstavljajo lokacije v spominu, kjer se podatki nahajajo:

prvo število na 1. lokaciji: 110110, drugo na 2. lokaciji: 111100 in 3. lokacija predstavlja mesto, kamor se shrani izračunan podatek. Tako bi lahko omenjeno seštevanje napisali v strojnem jeziku kot:

011110 110110 111100 111101

Na prvih računalnikih, ko še ni bilo višje nivojskih programskih jezikov, je bilo potrebno programirati na nivoju ničel in enic. Iskanje napak v takšnih programih je bilo zelo težavno.

Danes ne rabimo več skrbeti glede strojnega jezika, ker lahko uporabimo poseben program, da prevede višje-nivojsko ali **izvorno kodo** programa v strojni jezik ali objektno kodo, ki je edina koda, ki jo lahko izvršimo ali poženemo s pomočjo računalnika. V splošnem rečemo takšnemu programu prevajalec (angl. translator). Tako lahko z ustreznim prevajalcem za jezik Java ali C pišemo programe, kot da bi računalnik razumel ta jezik.

Prevajalci izvorne kode obstajajo v dveh različicah. **Tolmači** ali **interpreterji** (angl. interpreters) prevajajo vrstico za vrstico in izvršijo kodo vrstice preden prevedejo naslednjo vrstico. **Prevajalniki** (angl. compilers) pa prevedejo celoten program v izvršljiv program. Zaradi tega so učinkovitejši, odkrivanje napak pa je težje. Dandanes imamo vse več programskih jezikov (Java, Python), ki združujejo prednosti enih in drugih in tako interpretirajo izvorno kodo v vmesno kodo, ki jo potem prevedejo s pomočjo prevajalnikov.

Zbirni jezik

Leta 1950 se je programiralo v zbirnem jeziku. V primerjavi s strojnim jezikom je bilo to kar zadovoljivo programsko okolje. Ljudje, ki so programirali, so bili tehnično usmerjeni, poznali so

delovanje računalnika in so lahko s pisanjem programov prihranili čas izvajanja, kar je pri takratnih računalnikih veliko pomenili, saj so bili viri zelo omejeni.

V naslednjih desetletjih se je pojavila potreba, da bi tudi netehnični ljudje pisali programe. Pojavila se je potreba po višje-nivojskih jezikih. V istem času je tudi računalnik postal zmogljivejši kar je omogočilo, da je bilo daljše izvajanje sprejemljivo. Tudi računalniški viri niso bili več tako omejeni.

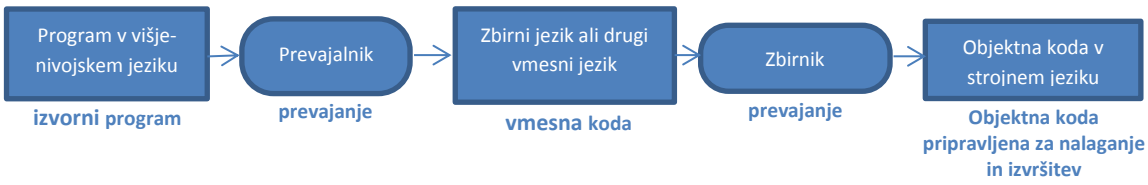
V zbirnem jeziku - zbirniku moramo paziti in poznati lokacijo v spominu. Vsak ukaz se prevede v strojni ukaz. Še več, program napisan za stroj X ne teče na drugem stroju, ki ima drugačne ukaze. Vse to so glavne pomanjkljivosti zbirnega jezika.

Visoko-nivojski jeziki odpravijo te pomanjkljivosti:

- Programer ne rabi skrbeti za nalaganje podatkov iz spomina oziroma vedeti, kje se ti podatki v spominu nahajajo.
- Ne rabi poznati nalog na najnižjem nivoju, ampak se lahko skoncentrira na reševanje problema na višjem nivoju, kot je v našem primeru sešteti dve števili v spremenljivko C.
- Programi so prenosljivi in niso odvisni od naprave.
- Izrazi so na nivoju naravnih jezikov in uporabljajo standardne matematične notacije.

Visoko-nivojske programske jezike imenujemo tudi jeziki tretje generacije, ki nakazujejo napredek iz strojnega jezika (prva generacija) v zbirni jezik (druga generacija). S tem smo dosegli nov nivo abstrakcije, ki nas še bolj oddalji od nizko nivojskih elektronskih komponent naprave.

Tako kot smo za prevod iz kode zbirnika v strojno kodo potrebovali zbirnik (angl. assembler), sedaj potrebujemo nov prevajalnik (angl. compiler), ki prevede izvorno kodo višjega programskega jezika v vmesno kodo. Ta vmesna koda se nato prevede s pomočjo zbirnika v objektno kodo.



Večina visoko-nivojskih jezikov je tako imenovanih **postopkovnih jezikov**. Za te jezike je značilno, da program sestavlja zaporedje ukazov, ki se izvršujejo in rešijo določeno nalogo. To sovpada z arhitekturo Von Neumanovega računalnika, ki opisuje zaporedje ciklov: prenos + izvajanje. Tako so tudi osnovne operacije postopkovnih jezikov predvsem shranjevanje in pridobivanje vrednosti. Različni jeziki se razlikujejo tako po pravilih, kako morajo biti stavki napisani (sintaksa oz. skladnja), kot tudi po pomenu pravilno napisanih stavkov (semantika).

Sintaksa programskega jezika enolično **določa obliko dovoljenih izrazov** v danem programskem jeziku. Če program ni napisan v skladu s sintaktičnimi pravili jezika, prevajalnik odkrije napake v programu in uporabnika z ustreznimi sporočili na to opozori.

Semantika programskega jezika enolično **določa interpretacijo (pomen) izrazov** v danem programskem jeziku. Na splošno prevajalnik ne more odkriti semantičnih napak v programu, saj ne ve, kaj je uporabnik s programom hotel opisati. Do določene mere je semantika določena s sintakso

programskega jezika. Semantične napake, vezane na sintakso jezika, lahko prevajalnik odkrije (npr. spreminjanje vrednosti konstante).

Namenski programski jeziki

Čeprav imajo omenjeni postopkovni programski jeziki (COBOL, FORTRAN, Pascal, C++, C#, JAVA, Python) močna področja, vsi veljajo za splošno namenske programske jezike. Poznamo pa tudi jezike, ki so namenjeni za določeno področje: podatkovne baze (SQL), spletne strani (HTML, JavaScript).

Druge paradigme programiranja

Paradigma postopkovnega programiranja pravi, da zaporedje ukazov posredujemo računalniku. Vsak ukaz dostopa ali spremeni vsebino v spominu računalnika. Če računalnik zaporedno izvaja ukaze, je končna rešitev zadnje stanje v spominu.

Pravzaprav programiranje sestoji iz dveh delov: najprej načrtovanje algoritma-postopka, potem pa zapis nedvoumnih operacij kot zaporedje ukazov. Pri postopkovnem načinu moramo na rešitev naloge gledati kot na reševanje **korak po koraku**. Tudi pri objektnem programiranju način ostaja enak, samo da so koraki porazdeljeni med posamezne podnaloge razredov.

Poznamo še druge načine programiranja – programiranje, ki bazira na drugih paradigmah. Tako bi lahko primerjali učenje postopkovnega jezika z učenjem nemščine, španščine , italijanščine (različni, a podobni jeziki) in se sedaj hočemo naučiti arabsko, japonsko ali znakovni jezik – jeziki ki so popolnoma drugačni po obliki, strukturi in abecedi.

Funkcijsko programiranje

Pri funkcijskem programiranju vsako nalogo opišemo s pomočjo funkcije. Tukaj je funkcija mišljena kot matematična funkcija: f(x)=2*x. Funkcija vzame argument ali več argumentov in vrne rezultat. Poznamo primitivne funkcije, ki so del jezika. Druge lahko napišemo sami. Pri klicanju funkcij velikokrat gnezdimo tudi funkcije v samo funkcijo, tako da je **rekurzivni način** prevladujoči način pri funkcijskem programiranju.

Logično programiranje

Pri funkcijskem programiranju se oddaljimo od implicitnega podajanja navodil za posamezen korak, ki ga mora računalnik narediti. Namesto tega določimo transformacije podatkov - funkcije in njihovo kombinacijo, ki nas pripelje do rešitve.

Pri logičnem programiranju gremo še korak naprej s tem, da ne podamo natančno, kako naj bo naloga rešena. Enostavno samo naštejemo **dejstva**, ki veljajo, in potem logični program sklepa nadaljnja. Logičnim programskim jezikom pravimo tudi deklarativni jeziki (v nasprotju z ukaznimi jeziki), ker v programe namesto ukazov zapisujemo veljavna dejstva.

Program sestavljajo **dejstva** in **pravila**. Programer zgolj pove dejstva in pravila določenega področja, ne potrebuje pa podajati navodil računalniku korak po koraku, kako pride do odgovora na določeno

povpraševanje. V nasprotju z ukaznimi jeziki pri deklarativnih jezikih opišemo KAJ naj program naredi in ne KAKO naj to naredi.

Paralelno programiranje

Čisto za konec omenimo še **paralelno programiranje**, ki dandanes, ko imamo večjedrne procesorje oziroma več računalnikov povezanih v mreže, pridobiva na pomenu. Naloga paralelnih programskih jezikov je čim bolj zaposliti vse procesne enote. Pojavlja pa se tudi vse večja težnja, da se paralelizem vključi v same prevajalnike, ki zaporedno napisan program sami porazdelijo med razpoložljive procesorske enote.

Programi kot zaporedja ukazov v sklopu tekmovanja BOBER

Večina nalog sega na področje postopkovnega programiranja – postopkovnih programskih jezikov. To se pravi, da računalniku posredujemo zaporedje ukazov. Potem pa računalnik zaporedno izvaja ukaze korak po koraku. Pri tem vsak ukaz spreminja stanje in končna rešitev je zadnje stanje.

Naloge iz področja programiranja lahko razdelimo v 4 skupine.

1. V prvo skupino spadajo naloge, ki **podajajo zaporedje ukazov**, ki mu moramo slediti, da rešimo nalogo. Slediti moramo torej programu – navodilom.

2. V drugo skupino spadajo naloge pri katerih moramo takšno **zaporedje ukazov zapisati** sami, da potem napisan program reši nalogo.

3. Pri bolj zahtevnih nalogah ukazi niso trivialni ali pa je naloga težja zaradi same razumevanje notacije. Tako lahko v tretjo skupino uvrstimo naloge, ki za razliko od prejšnjih skupin ne podajajo natančnega opisa ali zaporedja ukazov temveč samo **opisujejo postopek** iz katerega moramo sami razvozlati zaporedje ukazov.

4. V svojo skupino pa lahko uvrstimo naloge, ki zahtevajo od nas naprednejše sledenje postopku. Pri teh nalogah moramo razumeti algoritem, pravila igre in razmisliti o tem, kako ga optimalno uporabiti oziroma v okviru pravil igre **poiskati optimalno rešitev**. Pri teh nalogah ni dovolj, da najdemo rešitev ampak moramo poiskati najbolj optimalno rešitev.

V nadaljevanju bomo spoznali nekaj primerov takšnih nalog. Naloge so razdeljene po zgoraj opisanih skupinah.

1. Zaporedje ukazov

Naloge podajo **zaporedje ukazov**, ki mu moramo slediti, da rešimo nalogo.

Nalogo lahko rešimo, da poskušamo zapisati zaporedje ukazov, ki bo rešilo dani problem ali pa **sledimo ukazom** in si zapisujemo vmesna stanja med izvajanjem.

Imamo	PRESTAVI(C,B)	PRESTAVI(A,C)	PRESTAVI(A,B)		Končno stanje:
A:V,S,Z B: C:R,M,O	A:V,S,Z B:R C:M,O	A:S,Z B:R C:V,M,O	A:Z B:S,R C:V,M,O	X	A:Z B:V,R C:S,M,O

Imamo	PRESTAVI(C,B)	PRESTAVI(A,B)	PRESTAVI(A,C)		Končno stanje:
A:V,S,Z B: C:R,M,O	A:V,S,Z B:R C:M,O	A:S,Z B:V,R C:M,O	A:Z B:V,R C:S,M,O	=	A:Z B:V,R C:S,M,O

Imamo	PRESTAVI(A,B)	PRESTAVI(C,B)	PRESTAVI(A,C)		Končno stanje:
A:V,S,Z B: C:R,M,O	A:S,Z B:V C:R,M,O	A:S,Z B:R, V C:M,O	A:Z B:R,V C:S,M,O	X	A:Z B:V,R C:S,M,O

Imamo	PRESTAVI(B,C)	PRESTAVI(C,B)	PRESTAVI(A,B)		Končno mora:
A:V,S,Z B: C:R,M,O	A:V,S,Z B: C:R,M,O	A:V,S,Z B:R C:M,O	A:S,Z B:V,R C:M,O	X	A:Z B:V,R C:S,M,O

Razhroščevanje

Če ste že preizkušali programe, se vam je morda že kdaj zgodilo, da program ni naredil točno tistega, kar ste hoteli. To se dogaja pogosto. Pri razhroščevanju ugotavljamo kaj računalnik počne (korak po koraku) in kaj moramo storiti, da bo delal, kar bi hoteli mi. To je lahko precej zahtevno. (Iz lastnih izkušenj vam lahko povem, da lahko tudi najmanjšo napako iščemo cel teden ali več.)

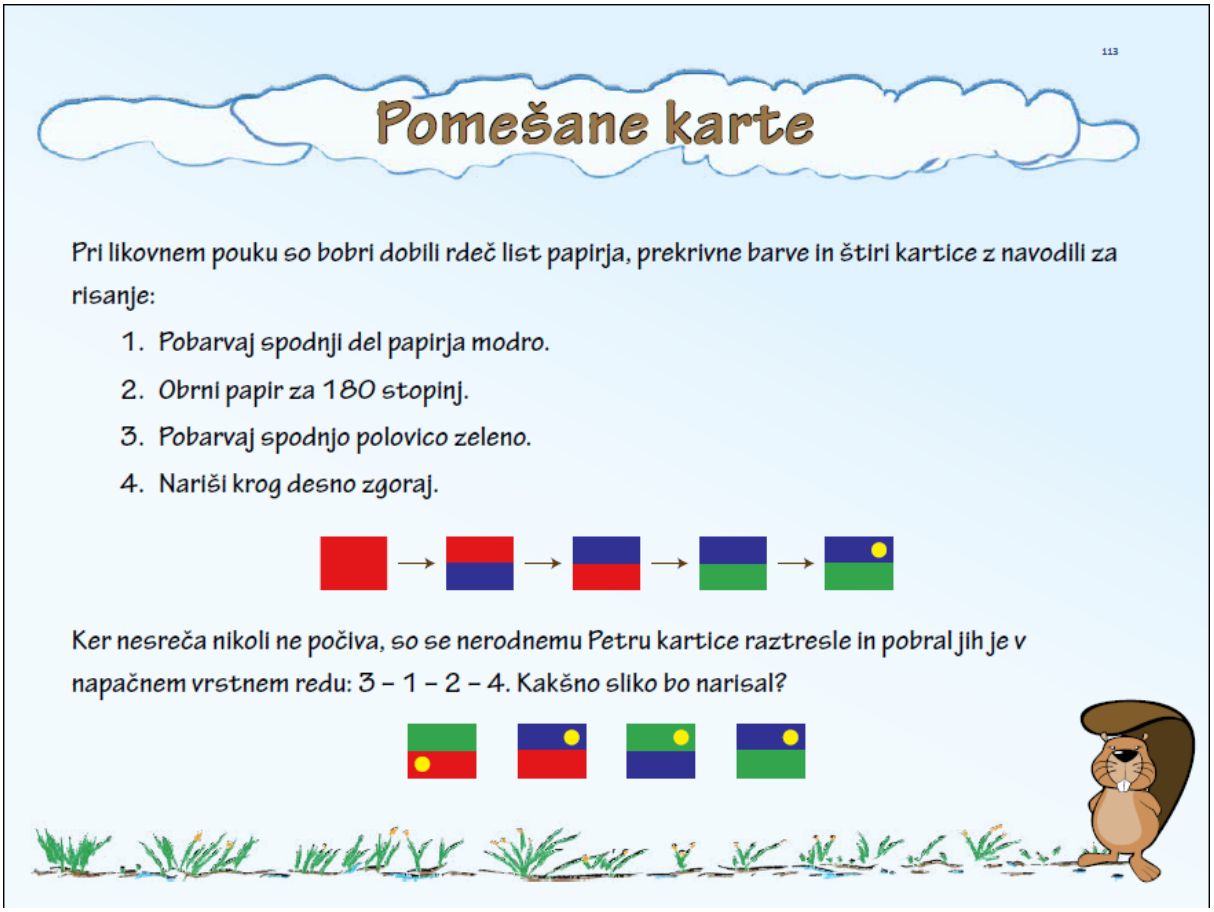
Podobne naloge: *Skladiščenje hlodov*, *Stroj za prestavljanje krožnikov* (sklad),

Kje je Franci?, *Jadranje*

Sledimo opisu poti iz vsake hiše. Za razliko od pa parkiranja imamo tukaj več možnih rešitev, zato sledimo postopku. Razlika je tudi, da tukaj nimamo ukaza naprej, ki določa premik do naslednjega križišča.

Bager

Pri tej nalogi sledimo ukazom. Ugotovimo, da nas več različnih postopkov pripelje do rešitve. Izbrati moramo najbolj optimalno.



Vrstni red ukazov

Tukaj imamo zopet seznam navodil – ukazov, ki mu moramo slediti, da izdelamo sliko. Enostavno sledimo postopku in končno stanje je narisana slika.

Kaj se lahko še naučimo iz omenjene naloge? Spoznamo, da je vrstni red ukazov lahko zelo pomemben. Pri nalogi smo videli, da je drugačen vrstni red pri pomešanih kartah dal drugačen končni rezultat – drugačno sliko.

Podobne naloge: *Pirhi, Slika iz štampiljk,*

Ugašanje (vrstni red ni pomemben)

Pri nalogi ponovno sledimo ukazom, ki izriše cvet. Tudi tukaj lahko vidimo, da je zaporedje ukazov zelo pomembno: rumena – oranžna – zelena – rdeča. Spoznamo pa še en koncept programiranja in to je ponavljanje.

Ponavljanje

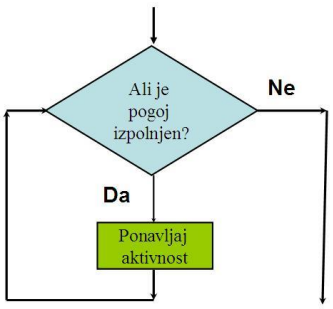
Velikokrat se srečamo z izvajanjem opravil, kjer moramo večkrat ponoviti isto aktivnost. Dober primer takšnega opravila je peka palačink. Za pripravo palačinke moramo najprej v posodo za peko naliti olje, nato v posodo nalijemo mešanico mleka, jajc in olja, palačinko nato pečemo najprej na eni strani, nato jo obrnemo in popečemo še na drugi strani. Ko je palačinka pečena, jo damo na krožnik, namažemo in zvijemo. Postopek peke palačink ponavljamo dokler nismo spekli želeno število palačink ali dokler nam ne zmanjka mase za palačinke.

Podobno opravilo je tudi prenašanje opek iz enega mesta na drugo. Takšen prenos opek lahko opišemo z naslednjim postopkom (psevdokodom):

dokler nisi preložil vse opeke
    vzami opeko
    prenesi opeko
    spusti opeko

V splošnem lahko vsako ponavljanje opravila zapišemo na način:

dokler (pogoj)
    stavek



To lahko razumemo kot: Dokler je pogoj izpolnjen: Izvajaj stavek. Ko pogoj ni več izpolnjen: Izvedi stavek, ki sledi strukturi ponavljanja. Običajno imamo na razpolago še zanko, ki ji podamo kolikokrat se ponovi (zanka **for**).

Podobne naloge: *Bobri predejo mrežo* (razumevanje notacije)

2. Zapis korakov

Nalogo moramo zapisati kot zaporedje korakov, ki reši dani problem: v tem primeru pripelje avto do parkirišča. Zaporedju ukazov lahko rečemo tudi program.

Pri tej nalogi moramo paziti, ker imamo tudi ukaz naprej (N), ki avto dejansko premakne do drugega ovinka.

Pravilen zapis ukazov: N, L, N, D, N, L, N,D, N

Iz zaporedja ukazov lahko ugotovimo, da se del ukazov ponavlja (podčrtan del). Tako, da lahko v okviru tega spregovorimo tudi o konceptu podprograma, ki je pri programiranju zelo pomemben.

Podprogrami

Če opazimo ponavljajoče bloke zaporedja ukazov v programu lahko program zapišemo z manj ukazi tako, da ponavljajoči blok zapišemo v tako imenovan podprogram. Ta podprogram nato kličemo iz glavnega programa. Tako bi lahko zgornji program zapisali kot:

MAIN: P1, P1, N

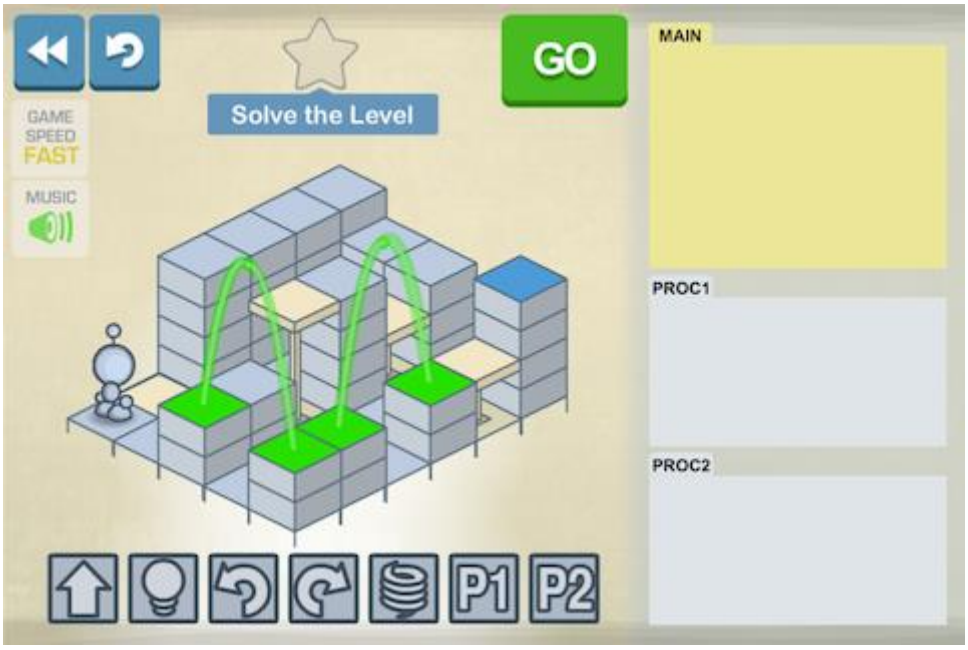
P1: N,L,N,D

Običajno podprogrami zaokrožujejo določeno nalogo. V različnih programskih jezikih jih imenujemo procedure, funkcije, rutine, podprogrami. Pri objektnem programiranju jih imenujemo metode in

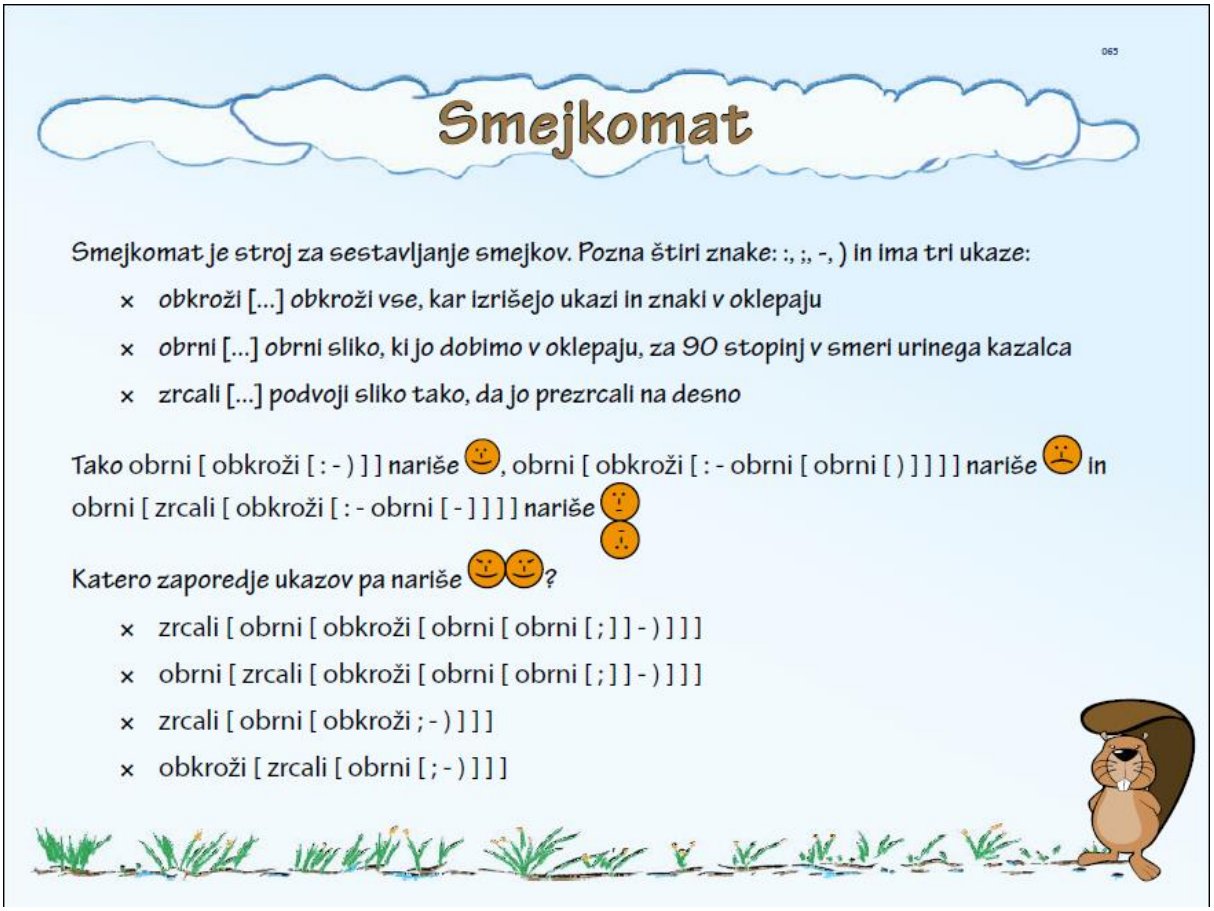
vezane na posamezne objekt. Lahko jih definiramo v programu ali pa v knjižnicah, tako da so na voljo tudi drugim programom. Podprogram je lahko napisan tako, da sprejme parametre – podatke iz glavnega programa in vrne vrednost glavnemu programu.

Podobne naloge, ki od nas zahtevajo zapis ukazov: *Žabin sprehod*, *Pot do cveta* (binarno drevo: levo ali desno)

Zanimivo: Igrica Light bot (http://light-bot.com/), ki nas uči programiranja ter spoznavanja konceptov, kot so podprogram, pogoji, zanke: Na voljo za iOS, Android in Flash.



3. Razumevanje opisa

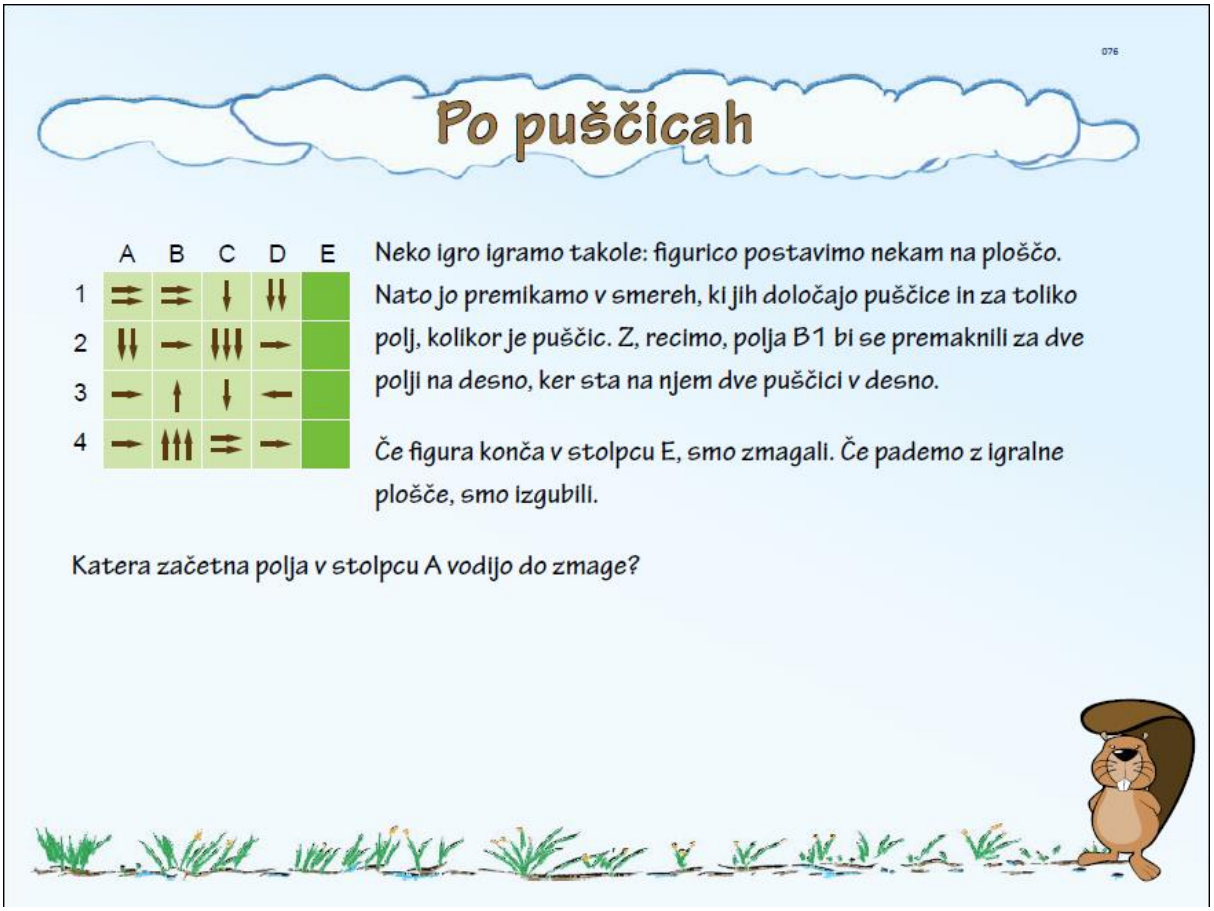


Funkcijsko programiranje

Pri tej nalogi imamo ravno tako ukaze samo, da so ti ugnezdeni. To nalogo lahko uvrstimo v 3 skupino, ker nimamo zapisanega pravega zaporedja ukazov – kot v postopkovnih programih ampak je program zapisan bolj v obliki funkcijskega programiranja. Znotraj posamezne funkcije imamo drugo funkcijo.

Podobne naloge: *Dvoštevila, Rože rastejo, Ujemi barvo*

4. Poiskati moramo rešitev



Sledimo postopku: Za vse kvadratke, ki še niso označeni:

Pot katera pripelje do cilja označimo, drugače pa celotno pot prečrtamo.

Če že pridemo na označeno pot končamo.

Lahko tudi obratno: Pogledamo kateri kvadratki nas pripeljejo na cilj in jih označimo. Potem preverimo za vse označene kvadratke, kateri kvadratki pripeljejo do njih.

Nalogo lahko začnemo s preizkušanjem in štejem, kateri poskus nam prinese najboljšo rešitev. V najslabšem primeru prestavimo vse vagone z obeh tirov na srednjega in vagone uredimo. Ampak za to bomo potrebovali 16 kovancev. Če smo malce bolj iznajdljivi in zadnji vagon na koncu prestavimo raje na drugi oz. prvi tir namesto na srednjega zapravimo 14 kovancev. Pri igri moramo vedno težiti k barvni urejenosti vagonov in k čim manjši uporabi srednjega tira.

Če pa hočemo nalogo zapisati s postopkom, da jo bo mogoče tudi računalnik znal rešiti pa moramo zapisati vse možne premike - nova stanja v obliki grafa in preiskovati po takšnem grafu, dokler ne najdemo najbolj ugodne rešitve.

Podobne naloge: *Zmeda na postaji (2), Preskakovanje, Labirint*

Grafi in iskanje v globino in širino

Vsako nalogo moramo prepisati v stanja. Običajno imamo vedno začetno stanje iz katerega hočemo doseči neko končno stanje. Pri vsaki nalogi imamo podana navodila, kako preidemo iz enega stanja v drugo. Če povežemo stanja, ki si sledijo ena v drugo dobimo povezan graf.

Iskanje rešitve običajno poteka tako, da začnemo v začetnem stanju in potem obiskujemo vmesna stanja, dokler ne pridemo do rešitve. Če pridemo do končnega stanja od koder ne moremo več naprej se vrnemo nivo višje in preiskujemo naslednjo vejo, dokler ne pridemo do rešitve. Celotna pot od začetka do rešitve je končna rešitev, ki nam da postopek.

Rešitev lahko iščemo na več načinov. Tako poznamo iskanje v globino, kjer v vsakem koraku pogledamo eno stanje naprej – se spuščamo vedno en nivo globlje.

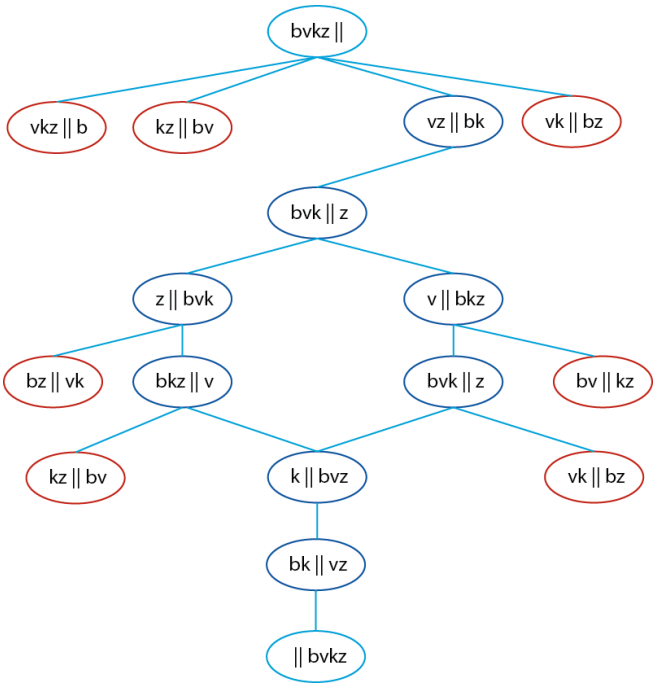
Pri preiskovanju v širino pa najprej pogledamo vsa možna stanja iz začetnega stanja nato pa gremo in pogledamo vsa stanja, ki so en nivo nižje in tako naprej.

Brodnikov problem

Poglejmo si opisana iskanja na dobro poznani uganki

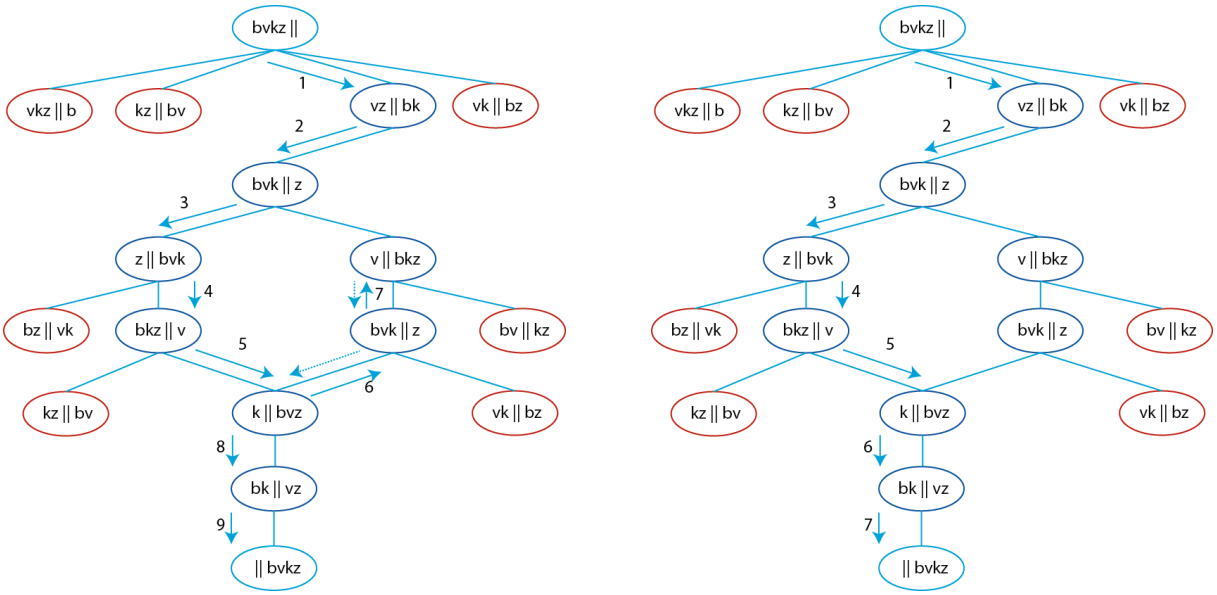
Brodnik mora na nasprotni breg reke spraviti kozo, volka in zelje. Pri tem lahko v čolnu pelje le eno od obeh živali ali le zelje. Poleg tega ne sme na istem bregu pustiti koze same z volkom ali koze same z zeljem.

Najprej moramo prepisati uganko v povezan graf možnih stanj (slika 1). Dve stanji bomo povezali, če lahko z enim prevozom preidemo iz tega stanja v drugo. Pri reševanju problema (brodnika, volka, koze in zelja = bvkz) bomo začeli z začetnim stanjem in poskušali doseči ciljno stanje (vsi na drugi stani reke) tako, da se bomo sprehodili po vmesnih stanjih. Stanja lahko zapišemo tako, da udeležence zapišemo v seznam prvih črk (bvkz ||), reko med njimi pa predstavimo z dvema navpičnicama. Cilj je vse pripeljati na drugo stran (|| bvkz).



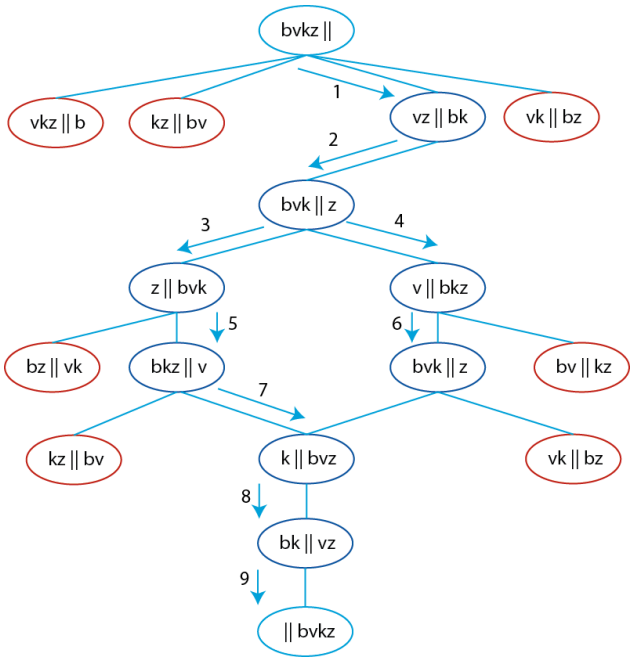
Slika 1: Graf možnih stanj za Brodnikov problem

Pri iskanju v globino začnemo z začetnim stanjem in nato v naslednjem dovoljenem (vz || bk). Pri preiskovanju si zapomnimo katera stanja smo že obiskali. Če pridemo, do konca in to še ni rešitev se vrnemo nazaj. Tako nadaljujejo z naslednjim stanjem, ko se brodnik vrne (bvk || z). Nadaljujemo v desni veji (z|| bvk) in tako naprej. Ko pridemo do stanja, ko imamo samo še kozo na levi strani (k||bvz) imamo dve možnosti navzdol ali pa levo. Za preiskovanje je to vse naslednik trenutnega stanja, ki ga še nismo obiskali. V primeru, če obiščemo stanje (bvk || z) se bomo pri stanju (v || bkz) morali vrniti, ker smo naslednje stanje že obiskali. Obe nakazani rešitvi prikazujeta sliki 2 in 3.



Sliki 2 in 3: Postopka preiskovanja v globino

Pri preiskovanju v širino pa najprej preiščemo vse naslednike. V našem primeru gremo iz stanja (bvk || z) najprej pogledati desno in nato še levo in tako naprej. Postopek iskanja prikazuje slika 4.



Slika 4: Preiskovanje v širino.

Boštjan Slivnik
Gramatike in avtomati

V pojasnilo in opravičilo hkrati naj takoj na začetku zapišem, da se kljub najboljšim namenom ni mogoče izgoniti kaki matematični formuli tu in tam. Taka je narava avtomatov, formalnih jezikov in gramatik. Zainteresiranega bralca to ne bi smelo motiti.

Avtomati in gramatike imajo za človeka z običajno "klasično" izobrazbo kaj malo skupnega. Slovnico spoznamo najprej pri učenju materinega jezika, kasneje pa nekaj malega tudi pri učenju tujih jezikov. O avtomatih običajno pri pouku ne slišimo kaj dosti, v šoli najpogosteje stojijo v kotu in so namenjeni kuhanju kave.

A nič ne de, na nekaj straneh bomo skušali predstaviti, kaj v teoretičnem računalništvu pomenijo besede avtomat, gramatika, formalni jezik. Preden pa se dokončno zakopljemo v težave, si najprej poglejmo zgodovinski potek razvoja tega področja.

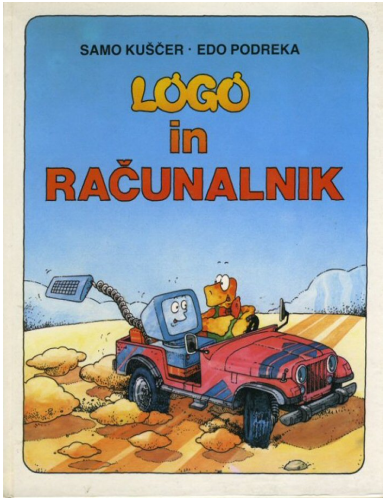
Morda lahko začnemo z Gottfriedom Leibnizem, nemškim matematikom, ki je že v sedemnajstem stoletju uspešno sestavil mehanski računski stroj, nato pa začel razmišljati o stroju, ki bi lahko s preurejanjem simbolov lahko ugotavljal pravilnost matematičnih trditev. Spoznal je, da bi za uporabo takega stroja moral obstajati nek formalen jezik za opis matematičnih trditev, in v nadaljevanju svojega dela se je zato posvetil prav temu. Neka ohlapna zveza med stroji in jeziki, med avtomati in gramatikami, se je očitno rodila že zelo zgodaj.

Precej kasneje, leta 1928, je nemški matematik David Hilbert idejo o ugotavljanju pravilnosti matematičnih trditev formalno opisal kot "odločitveni problem" oziroma po nemško "Entscheidungsproblem". Rešitev tega problema bi bil nek natančen postopek, ki bi za vsako matematično izjavo v logiki prvega reda izračunal, ali je ta izjava vedno resnična ali ne. Vhod v algoritem bi bila torej izjava v logiki prvega reda, izhod pa "da" ali "ne", pač glede na opisano lastnost vhodne izjave.

Bodimo pozorni na to, da je Hilbert zahteval zgolj natančen postopek in ne algortima, saj pojem algoritma leta 1928 sploh še ni bil formalno definiran. To je komaj nekaj let kasneje, leta 1936, uspelo ameriškemu matematiku Alonzu Churchu in angleškemu matematiku Alanu Turingu. Church je definicijo algoritma zasnoval v okviru svojega posebej za rešitev "Entscheidungsproblem"-a razvitega λ-računa, Turing pa na osnovi svojega računskega modela, ki se danes po njem imenuje Turingov stroj. Mimogrede, odličen uvod v računalništvo za osnovnošolce je slikanica Sama Kuščerja, ki predstavi Turingov stroj kot "toaletni računalnik" — zakaj, naj ostane skrivnost kot vzpodbuda, da bralec sam poseže po tej knjigi.

Samo Kuščer, Logo in računalnik,
Velika izobraževalna slikanica,
ilustr. Edo Podreka, Mladinska knjiga, 1987:

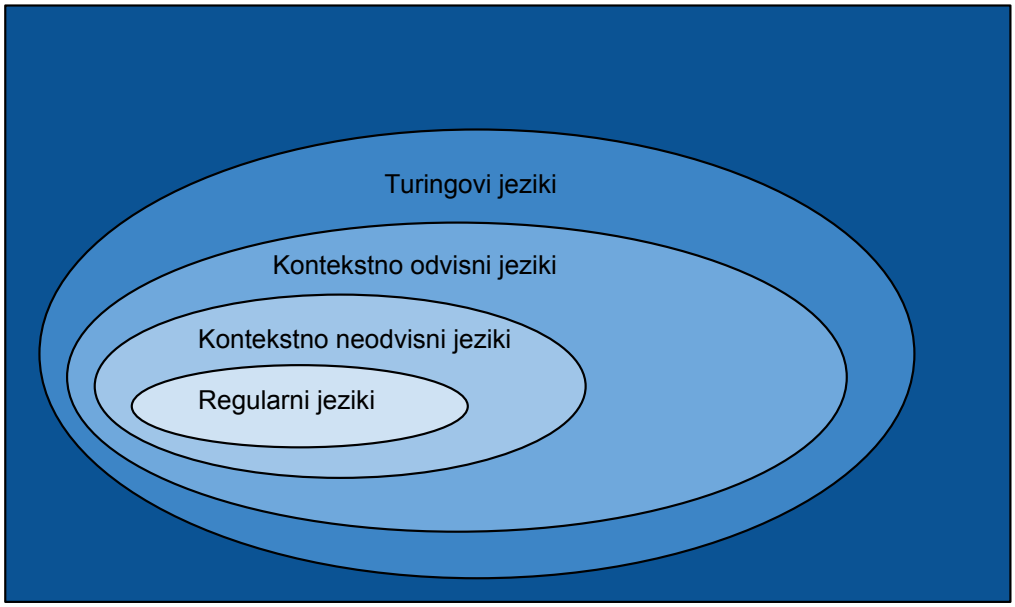
Odlična knjiga za osnovnošolce, ki na lahkoten in zabaven način predstavi delovanje in programiranje današnjih računalnikov.



A hkrati z definicijo algoritma je prišel tudi za tiste čase razmeroma presenetljiv odgovor na "Entscheidungsproblem": rešitve ni! Ko je Hilbert problem definiral, ni niti pomislil na to, da bi morda lahko bil nerešljiv. A Church je dokazal, da ne obstaja algoritem, ki bi ugotovil, ali sta izraza v λ-računu ekvivalentna. Turing je v istem času, le na drugi strani Atlantika, dokazal, da noben Turingov stroj ne more preveriti, ali nek drug Turingov stroj zasnovan tako, da se bo ne glede na vhodne podatke izračun na tem stroju končal v sicer poljubno velikem, a kljub vsemu končnem številu korakov — ta problem je postal znan kot "problem ustavljivosti Turingovega stroja". Oba problema, Churcheva ekvivalenca dveh λ-izrazov in Turingov problem ustavljivosti sta povsem formalno definirana, a nerešljiva z algoritmom, kar pomeni, da postopka za rešitev "Entscheidungsproblem"-a ni.

V formalni dokaz nerešljivosti "Entscheidungsproblem"-a se na tem mestu ne bomo spuščali, saj presega obseg tega besedila in predavanj, ki mu je to besedilo namenjeno. Poleg tega pa oba dokaza, Churchev in Turingov, temeljita na ugotovitvah nemškega matematika Georga Kantorja, ki se mnogim "klasično" izobraženim ljudem na prvi pogled zdijo vsaj nekoliko čudaške. Recimo, da je lihi celih števil enako mnogo kot vseh celih števil skupaj, da je vseh celih števil enako mnogo kot vseh ulomkov, da pa je vseh celih števil manj kot vseh realnih števil. Še več, da obstaja neskončno mnogo različnih neskončnosti. In verjemite, dokaz sploh ni tako zapleten, en sam list papirja zadostuje — le globoko vkoreninjene predstave o "krompirju-u-gajbi" je treba biti pripravljen še enkrat premisliti.

Čeprav sta oba delala povsem neodvisno, se je izkazalo, da sta oba modela računanja, Churchev λ-račun in Turingov stroj, natančno enako močna. Kar lahko izračunamo z enim, lahko tudi z drugim; in obratno. Celo še več: vsi modeli računanja, ki so jih definirali kasneje in ki naj bi bili kar se da močni, hkrati pa bi se jih dalo fizično narediti, so se izkazali za natanko enako močne — vključno z najzmogljivejšim elektronskim superračunalnikom. In pri tem ne pozabimo, da sta Church in Turing definirala vsak svoj modela računanj v času, ko računalnikov še ni bilo.



Slika 1: Chomskyjeva hiearhija formalnih jezikov.

A tako kot sta Church in Turing vsak zase neodvisno definirala pojem algoritma in podala razlago "Entscheidungsproblem"-a, tako se je v drugi polovici dvajsetega stoletja znanje o načinih in mejah formalnega računanja razvijalo po dveh navidez precej ločenih, a v bistvu tesno povezanih poteh.

Po eni poti so kljub dejstvu, da "Entscheidungsproblem" ni rešljiv, ali pa morda prav zaradi tega, so mnogi matematiki in kasneje računalnikarji začeli raziskovati, kakšne probleme je vendarle mogoče reševati s formalnimi postopki računanja oziroma z računalniki. V drugi polovici 20. stoletja je bilo definiranih mnogo modelov računanja različne računske moči, pač glede na trenutne potrebe in možnosti fizične izdelave.

Po drugi poti so se matematiki in jezikoslovci lotili raziskovanja podobnih problemov. Najbolj znan raziskovalec na tem področju je gotovo Noam Chomsky, ki je uvedel idejo univerzalne slovnice, v okviru teh raziskav pa se je razvila teorija formalnih jezikov, ki je ustrezala tako jezikoslovcem kot računalnikarjem.

S stališča teoretičnega računalništva je gotovo najpomembnejši rezultat tega razvoja Chomskyjeva hierarhija formalnih jezikov. Ta definira štiri pomembne razrede formalnih jezikov, kot je to prikazano na sliki 1: regularne jezike, kontekstno neodvisne jezike, kontekstno odvisne jezike in Turingove jezike. Diagram na sliki 1 je predstavljen z množicami: množica vseh regularnih jezikov je prava podmnožica kontekstno neodvisnih jezikov, množica kontekstno neodvisnih jezikov je prava podmnožica kontekstno odvisnih jezikov, in tako naprej. Zunanji pravokotnik na sliki 1 predstavlja univerzalno množico vseh jezikov.

Jezikoslovci so za vsak razred jezikov Chomskyjeve hierarhije definirali razred formalnih gramatik, računalnikarji pa abstraktni matematični model računanja, ki ustreza vsakemu od teh jezikov v Chomskyjevi hierarhiji. Natančneje, Chomskyjevo hierarhijo lahko predstavimo tudi takole:

Turingovi jeziki:
gramatike brez omejitev
Turingovi stroji

Kontekstno odvisni jeziki:
kontekstno odvisne gramatike
linearno omejeni avtomati

Kontekstno neodvisni jeziki:
kontekstno neodvisne gramatike
skladovni avtomati

Regularni jeziki:
linearne gramatike
končni avtomati

V zgornji predstavitvi je za vsak razred najprej naveden razred gramatik, nato pa abstraktni matematični model računanja. Kot je razvidno iz sheme, obstaja zelo lepa povezava med opisom jezika z gramatiko in abstraktnim strojem (avtomatom).

Ne glede na poljudno naravo tega besedila je počasi nujno, da pojme formalnega jezika, gramatike in avtomata določimo natančneje.

Formalni jezik je množica besed končne dolžine, ki so sestavljene iz *simbolov* neke izbrane končne množice simbolov, ki ji rečemo *abeceda*.

Primer 1 Jezik vseh besed dolžine 3 nad abecedo $\{a, b\}$ je množica

$$\{aaa, aab, aba, abb, baa, bab, bba, bbb\} .$$

Jezik vseh besed nad abecedo $\{a, b\}$, ki vsebujejo enako število a-jev in b-jev, je množica

$$\{\varepsilon, ab, ba, aabb, abab, abba, baab, baba, bbaa, \ldots\} ,$$

pri čemer ε opisuje prazno besedo (torej besedo dolžine 0). Slednji jezik je neskončen, zato smo lahko zapisali le nekaj besed tega jezika. Bistroumni bralci bodo razumeli, kakšne so tudi ostale besede tega jezika, ostali pa naj se potrudijo in izpišejo še vse ostale, ki smo jih v zgornjem zapisu izpustili. ———

Malce poenostavljeno povedano je *gramatika* sistem prepisovalnih pravil, s katerimi iz enega niza simbolov izpeljemo drug niz simbolov. Pravilom pravimo *produkcije*, vsaka produkcija pa

ima levo in desno stran: če v nekem nizu simbolov najdemo podniz, ki ustreza levi strani neke produkcije, lahko ta podniz nadomestimo z nizom simbolov na desni strani produkcije. Če lahko v nekem trenutku uporabimo več produkcij, pač glede na trenuten niz simbolov in leve strani produkcij, potem lahko svobodno izberemo produkcijo in jo uporabimo.

Jezik gramatike sestavljajo vse besede, ki se jih da z ravnokar opisanim načinom uporabe produkcij izpeljati iz nekega vnaprej določenega simbola gramatike.

Primer 2 Vzemimo abecedo $\{a, b\}$ in gramatiko s petimi produkcijami

$$S \longrightarrow \varepsilon \quad S \longrightarrow aAS\,, \quad aA \longrightarrow Aa\,, \quad Aa \longrightarrow bAb \quad \text{in} \quad A \longrightarrow bAb\,.$$

Ob dogovoru, da je S začetni simbol te gramatike, gramatika omogoča mnogo različnih izpeljav, med drugim naslednje tri:

$$S \Longrightarrow \varepsilon$$

$$S \Longrightarrow aAS \Longrightarrow AaS \Longrightarrow baS \Longrightarrow ba$$

$$S \Longrightarrow aAS \Longrightarrow aAaAS \Longrightarrow aAaA \Longrightarrow bAbA \Longrightarrow bbbA \Longrightarrow bbbb$$

Na osnovi teh treh izpeljav lahko zaključimo, da besede ε, ba in $bbbb$ pripadajo jeziku gornje gramatike. Pozorni bralec naj preveri, zakaj so to veljavne izpeljave, in nato še sam zapiše kakšno veljavno izpeljavo. ———————————————————

Gramatika torej *generira* jezik: z vsako izpeljavo, pri kateri uporabljamo produkcije gramatike, izpeljemo neko besedo jezika. Kot smo povedali že pri opisu Chomskyjeve hierarhije, obstaja več vrst gramatik:

1. *Gramatike brez omejitev:* pri teh gramatikah sta lahko na levi in na desni strani produkcije povsem poljubna niz simbolov, le niz na levi strani ne sme biti prazen.

2. *Kontekstno odvisne gramatike:* pri teh gramatikah mora biti niz na desni strani produkcije vedno vsaj tako dolg kot niz na levi strani produkcije.

   Ime "kontekstno odvisne gramatike" izvira iz dejstva, da se da vsako tako gramatiko pretvoriti v njej ekvivalentno gramatiko (ekvivalentno v tem smislu, da generira isti jezik), pri kateri so vse produckije oblike $xAy \longrightarrow x\omega y$, pri čemer sta x in y niza simbolov jezika generiranega jezika, ω pa je niz, ki ga lahko sestavljajo poljubni simboli gramatike (torej tako tisti, ki so v abecedi jezika, kot tisti, ki niso). Na tak način lahko simbol A zamenjamo z nizom ω le v natančno določenem kontekstu, ki ga določata niza x in y.

3. *Kontekstno neodvisne gramatike:* pri teh gramatikah mora biti na levi strani produkcije vedno natanko en simbol gramatike, ki pa ne sme biti simbol abecede jezika, na desni strani produkcije pa je lahko poljuben niz simbolov gramatike.

Ime "kontekstno neodvisna gramatika" izvira iz dejstva, da so vse produkcije oblike $A \longrightarrow \omega$. To pomeni, da sta niza x in y (glede na obliko produkcij kontekstno odvisnih gramatik) prazna in da uporaba produkcij torej ni odvisna od konteksta.

4. *Linearne gramatike:* pri teh gramatikah mora biti na levi strani produkcije vedno natanko en simbol gramatike, ki pa ne sme biti simbol abecede jezika, na desni strani produkcije pa je lahko niz simbolov abecede jezika, ki ima lahko (ali pa ne) na skrajni desni (levi) natanko en simbol gramatike, ki ni simbol abecede jezika.

Različne abstraktne modele računanja, ki se pojavljajo v Chomskyjevi hierarhiji, je precej težje predstaviti enotno (tako kot smo ravnokar predstavili gramatike). Linearno omejene avotmate in Turingove stroje bomo pustili za kakšno drugo priložnost, na tem mustu bomo predstavili le končne in skladovne avtomate:

1. *Končni avtomat* je model, ki ima končno število stanj, med katerimi prehaja glede na posamezne simbole trenutne vhodne besede. Med stanji je natančno eno stanje začetno, vsaj eno stanje pa mora biti označeno kot končno stanje.

   Za vsako podano besedo končni avtomat začne v začetnem stanju, nato pa v končnem številu korakov odgovori na vprašanje, ali ta vhodna beseda pripada jeziku tega avtomata. Na vsakem koraku avtomat na podlagi trenutnega stanja in trenutnega vhodnega simbola preide v novo stanje, trenutni vhodne simbol pa zavrže. Prehodi med stanji so vedno določeni glede na simbole vhodne besede, le v določenih korakih obstaja možnost, da vhodnega simbola za odločitev o prehodu iz enega stanje v drugega ne potrebuje in ga zato pri izvedbi koraka tudi ne zavrže (tak prehod je označen z ε). Natančneje, in morda bolj jasno, bomo delovanje končnega avtomata pokazali s primerom.

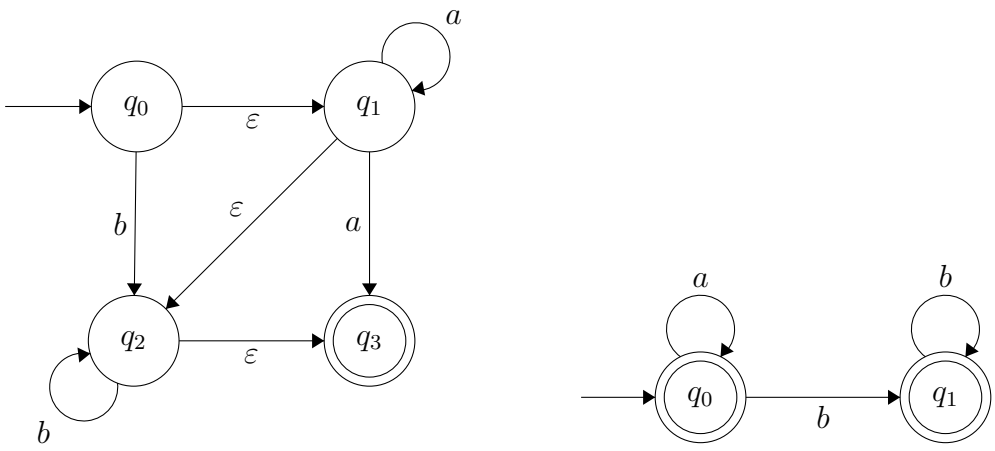
2. *Skladovni avtomat* je model, ki ima poleg končnega števila stanj še neskončno dol sklad, na katerega si lahko med izračunom shranjuje za sklad posebej določene simbole in si na ta način pomaga pri izračunu — sklad na nek način uporablja kot papir za "pomožne račune".

   Skladovni avtomat deluje podobno kot končni avtomat, le da je odločitev o prehodu v naslednje stanje odvisna tudi od simbola na vrhu sklada. Na vsakem koraku torej poleg trenutnega stanja in trenutnega vhodnega simbola za prehod v novo stanje uporabi tudi simbol na vrhu sklada, ki pa ga zato pri vsakem prehodu na vrhu sklada nadomesti z novim nizom skladovnih simbolov (ta niz je lahko seveda tudi prazen).

Da bi si bolje predstavljali prva dva nivoja Chomskyjeve hierarhije, predvsem pa delovanje končnih avtomatov, si oglejmo dva primera.

Primer 3 Najprej si oglejmo primer regularnega jezika. Vzemimo jezik

$$L_{a^n b^m} = \{a^n b^m \mid n \geq 0 \wedge m \geq 0\} \quad ,$$



Slika 2: Nedeterministični in deterministični končni avtomat za jezik $L_{a^n b^m}$.

ki ga sestavljajo vse besede, ki so sestavljene iz poljubnega števila a-jev in b-jev, pri čemer morajo vsi a-ji stati pred vsemi b-ji. Torej:

$$L_{a^n b^m} = \{\varepsilon, a, b, aa, ab, bb, aaa, aab, abb, bbb, \ldots\} \quad .$$

Ker je ta jezik regularen, zanj obstajata vsaj dva različna avtomata, ki sta prikazana na sliki 2. Pri obeh avtomatih se držimo dogovora, da začetno stanje označimo s prazno puščico, ki ne izvira iz nobenega drugega stanja, končna stanja pa označimo z dvojnim krogcem.

Delovanje prvega (na sliki 2 levo) lahko razložimo s primerom, ko mu na vhod damo besedo abb:

> Avtomat začne v svojem začentem stanju, torej v stanju q_0. Prvi simbol vhodne besede je a, zato je edina možnost, da avtomat brez uporabe tega simbola preide v stanje q_1. V stanju q_1 ima avtomat kar tri možnosti: (1) lahko uporabi simbol a in preide (ostane) v q_1, (2) lahko uporabi simbol a in preide v q_3 ali pa (3) simbola a ne uporabi in preide v q_2.
>
> Če izberemo prvo možnost, simbol a umaknemo, nov trenutni simbol postane (prvi) b, zato moramo v naslednjem koraku najprej opraviti prehod v stanje q_2 brez uporabe vhodnega simbola. Nato dvakrat opravimo prehod v q_2 po prvem in drugem simbolu b, nazadnje pa še prehod brez vhodnega simbola v končno stanje q_3. Ker smo končno stanje dosegli, zaključimo, da avtomat sprejme besedo abb oziroma da beseda pripada jeziku tega avtomata.
>
> Če pa bi izbrali drugo možnost, namreč prehod po vhodnem simbolu a v stanje q_3, procesa ne bi mogli nadaljevati, saj iz stanja q_3 ni prehoda po simbolu b, ki sledi a-ju. Ne glede na to, da smo dosegli končno stanje, na osnovi druge možnosti ne moremo zaključiti, da beseda abb priprada jeziku tega avtomata, saj avtomat ni obdelal celotne besede.

Opisana postopka lahko zgoščeno opišemo z naslednjima izpeljavama:

$$q_0abb \Longrightarrow q_1abb \Longrightarrow aq_1bb \Longrightarrow aq_2bb \Longrightarrow abq_2b \Longrightarrow abbq_3 \Longrightarrow abb \in L_{a^nb^m}$$

$$q_0abb \Longrightarrow q_1abb \Longrightarrow aq_3bb \Longrightarrow ???$$

Posebej bodimo pozorni na to, da slednja izpeljava ne zagotavlja, da beseda abb ni v jeziku $L_{a^nb^m}$ — je le zgrešen poskus, s katerim nam ni uspelo pokazati, da ta beseda ni v tem jeziku.

Avtomat na sliki 2 levo zahteva, da v določenih korakih izpeljave preprosto uganemo pravo pot — takemu končnemu avtomatu pravimo *nedeterministični končni avtomat*. Kot smo videli ravnokar, tak avtomat za nekatere besede dopušča več različnih začetkov izpeljave, od katerih nas vsi ne pripeljejo do končnega stanja. A velja celo še več, za nekatere besede obstaja celo več različnih izpeljav:

$$q_0aa \Longrightarrow q_1aa \Longrightarrow aq_1a \Longrightarrow aaq_3 \Longrightarrow aa \in L_{a^nb^m}$$

$$q_0aa \Longrightarrow q_1aa \Longrightarrow aq_1a \Longrightarrow aaq_1 \Longrightarrow aaq_2 \Longrightarrow aaq_3 \Longrightarrow aa \in L_{a^nb^m}$$

Da bi se izognili poskušanju pri preverjanju pripadnosti besede jeziku avtomata, lahko nedeterministični avtomat na live strani slike pretvorimo v *deterministični končni avtomat*, ki sprejema isti jezik, a v nobenem trenutku ne dopušča proste izbire v naslednjem koraku: iz vsakega stanja je za vsak vhodni simbol na voljo kvečjemu en prehod v neko novo stanje. Bralec naj za vajo skuša utemeljiti, da oba avtomata na sliki 2 res sprejemata isti jezik.

A ker je jezik $L_{a^nb^m}$ regularen, zanj obstajata desno linearna gramatika

$$S \longrightarrow A\,, \quad A \longrightarrow aA \,|\, B\,, \quad B \longrightarrow bB \,|\, \varepsilon$$

in levo linearna gramatika

$$S \longrightarrow B\,, \quad A \longrightarrow Aa \,|\, \varepsilon\,, \quad B \longrightarrow Bb \,|\, A\,.$$

Takoj lahko zapišemo izpeljavi

$$S \Longrightarrow A \Longrightarrow aA \Longrightarrow aB \Longrightarrow abB \Longrightarrow abbB \Longrightarrow abb$$

$$S \Longrightarrow B \Longrightarrow Bb \Longrightarrow Bbb \Longrightarrow Abb \Longrightarrow Aabb \Longrightarrow abb$$

ostale pa naj si bralec za vajo izpiše sam. ———————————————————

Primer 4 En nivo višje v Chomskyjevi hierarhiji so kontekstno neodvisni jeziki. Vzemimo jezik

$$L_{a^nb^n} = \{a^nb^n \mid n \geq 0\} \quad ,$$

ki je zelo podoben jeziku L_1 iz primera 3, le da morajo biti tu v vsaki besedi natančno enako število a-jev in b-jev. Torej:

$$L_{a^nb^n} = \{\varepsilon, ab, aabb, aaabbb, aaaabbbb, \ldots\} \quad .$$

Ker je jezik konteksnto neodvisen, zanj obstaja kontekstno neodvisna gramatika

$$S \longrightarrow aSb \,|\, \varepsilon \,.$$

Da ta gramatika res generira jezik $L_{a^n b^n}$, nam na prvi pogled zadostuje vzorec prvih nekaj izpeljav

$$\begin{aligned}
&S \Longrightarrow \varepsilon \\
&S \Longrightarrow aSb \Longrightarrow ab \\
&S \Longrightarrow aSb \Longrightarrow aaSbb \Longrightarrow aabb \\
&S \Longrightarrow aSb \Longrightarrow aaSbb \Longrightarrow aaaSbbb \Longrightarrow aaabbb \\
&\quad \vdots
\end{aligned}$$

vse preostale pa si lahko nejeverni bralec izpiše sam. ———————————————

Kot smo že povedali, so vsi regularni jeziki tudi kontekstno neodvisni, obratno pa seveda ni res. Kdor ne verjame, naj poskusi sestaviti končni avtomat za jezik $L_{a^n b^n}$ iz primera 4.

Na tem mestu moramo omeniti še *regularne izraze*, ki predstavljajo izredno učinkovit opis regularnih jezikov. Vsak regularni jezik lahko opišemo z nekim regularnim izrazom, ki ga dobimo na osnovi naslednjih treh pravil:

1. Regularni izrazi $\emptyset$, ε in a, pri čemer je a poljuben simbol abecede, opisujejo regularne jezike $\{\}$, $\{\varepsilon\}$ in $\{a\}$, zaporedoma.

2. Naj bosta r_1 in r_2 regularna izraza. Tedaj regularni izraz $r_1|r_2$ opisuje unijo jezikov regularnih izrazov r_1 in r_2, regularni izraz $r_1 r_2$ pa stik jezikov regularnih izrazov r_1 in r_2.

   V jeziku regularnega izraza $r_1|r_2$ je torej vsaka beseda, ki je v jeziku vsaj enega od obeh jezikov izrazov r_1 in r_2, v jeziku regularnega izraza $r_1 r_2$ pa je vsaka beseda, ki jo dobimo tako, da staknemo eno besedo iz jezika prvega regularnega izraza z besedo iz jezika drugega regularnega izraza.

3. Naj bo r regularni izraz. Tedaj jeziku regularnega izraza r^* pripadajo besede, ki nastanejo tako, da iz jezika regularnega izraza r vzamemo poljubno končno mnogo besed (lahko tudi nič; pri tem izboru se besede lahko tudi ponavljajo) in jih staknemo skupaj.

Dogovorimo se še, da operator * veže najmočneje, operator $|$ pa najšibkeje. Če želimo drugače, pač uporabimo oklepaje.

Primer 5 Ponovno izberimo abecedo $\{a, b\}$. Regularni jezik $L_{a^n b^m}$ lahko opišemo z regularnim izrazom

$$a^* b^* \, .$$

Zakaj? Izraz a opisuje jezik $\{a\}$, zato regularni izraz a^* opisuje jezik $\{\varepsilon, a, aa, aaa, aaaa, \ldots\}$. Po analogiji izraz b^* opisuje jezik $\{\varepsilon, b, bb, bbb, bbbb, \ldots\}$, stik teh dveh jezikov, kot ga opisuje regularni izraz $a^* b^*$ pa zato vsebuje besede, ki jih dobimo tako, da vzamemo neko besedo iz jezika $\{\varepsilon, a, aa, aaa, aaaa, \ldots\}$ in jo staknemo z neko besedo jezika $\{\varepsilon, b, bb, bbb, bbbb, \ldots\}$. Dobimo recimo besede $\varepsilon = \varepsilon\ \varepsilon$, $a = a\ \varepsilon$, $ab = a\ b$, $aaab = aaa\ b$, $\ldots$

Zapišimo še nekaj regularnih izrazov in jezikov, ki jih ti izrazi opisujejo:

$$\begin{aligned} a^* b a^* \quad &\ldots \quad \{b, aba, abaa, aaba, aabaa, \ldots\} \\ a^* (ba)^* \quad &\ldots \quad \{\varepsilon, a, ba, aa, aaba, ababa, baba, \ldots\} \\ (abb^* a)|bb \quad &\ldots \quad \{bb, aba, abba, abbba, \ldots\} \end{aligned}$$

Bralec naj se potrudi in poišče razlago sam.

Da pa ne bi ostalo vse skupaj samo pri teoriji, se vprašajmo, kje lahko formalne jezike, predvsem regularne in kontekstno neodvisne, tudi zares uporabimo.

Običajnemu programerju so gotovo najbližji regularni izrazi, saj jih pri programiranju zelo pogosto uporabljamo. Ne samo, da vsi uporabni urejevalniki besedil (ne, MS Word ne sodi v to skupino) omogočajo iskanje besed z uporabo regularnih izrazov, tudi v programih samih uporabljamo regularne izraze za iskanje podatkov v besedilih in podobnih zbirkah podatkov. Danes praktični ni več nobenega jezika, ki bi ne podpiral dela z regularnimi izrazi — bodisi so regularni izrazi del jezika bodisi del standardne knjižnice.

Primer 6 Recimo, da iščemo v neki datoteki vse vrstice, ki jih vsebujejo katerokoli letnico med 1980 in 2000. To lahko opišemo z regularnim izrazom

$$(19(9|8)(0|1|2|3|4|5|6|7|8|9))|2000$$

na računalniku pa z ukazom

```
$ grep "(19[89][0-9])|(2000)" datoteka
```

Na računalniku so regularni izrazi zapisani nekoliko drugače, saj so pri programiranju obogateni z večjim številom operatorjev. Med posameznimi izvedbami regularnih izrazov obstajajo določene razlike, zato je v vsakem primeru bolje pogledati na Google, kaj nam posamezno orodje ali posamezni programski jezik nudi v zvezi z regularnimi izrazi.

```
statement
        : labeled_statement
        | compound_statement
        | expression_statement
        | selection_statement
        | iteration_statement
        | jump_statement
        ;

labeled_statement
        : IDENTIFIER ':' statement
        | CASE constant_expression ':' statement
        | DEFAULT ':' statement
        ;

compound_statement
        : '{' '}'
        | '{' statement_list '}'
        | '{' declaration_list '}'
        | '{' declaration_list statement_list '}'
        ;

declaration_list
        : declaration
        | declaration_list declaration
        ;

statement_list
        : statement
        | statement_list statement
        ;

expression_statement
        : ';'
        | expression ';'
        ;

selection_statement
        : IF '(' expression ')' statement
        | IF '(' expression ')' statement ELSE statement
        | SWITCH '(' expression ')' statement
        ;

iteration_statement
        : WHILE '(' expression ')' statement
        | DO statement WHILE '(' expression ')' ';'
        | FOR '(' expression_statement expression_statement ')' statement
        | FOR '(' expression_statement expression_statement expression ')' statement
        ;

jump_statement
        : GOTO IDENTIFIER ';'
        | CONTINUE ';'
        | BREAK ';'
        | RETURN ';'
        | RETURN expression ';'
        ;
```

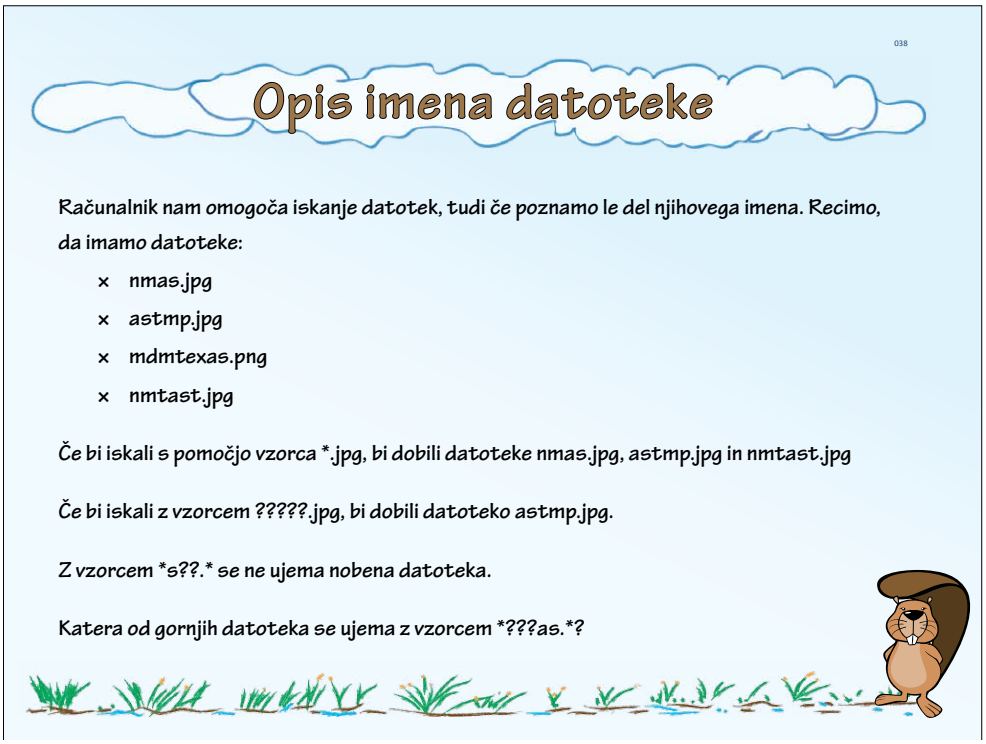
Slika 3: Del kontekstno neodvisna gramatike jezika C.

Končni avtomati so neposredno uporabni v računalniških komunikacijah, saj z njimi opisujemo protokole, uporabljamo jih tako pri snovanju logičnih vezij kot pri načrtovanju telefonskih central, pri sestavljanju modelov delovanja posameznih enot programa, ...

Kar se tiče kontekstno neodvisnih jezikov in gramatik, lahko zapišemo, se morda ne pojavljajo na toliko različnih področij računalništva, so pa zato povsem nepogrešljive za opis sintakse programskih jezikov in s prevajanjem programskih jezikov. A to je veliko: praktično ni področja računalništva, ki ne bi temeljilo na programiranju, za programiranje pa seveda potrebujemo prevajalnike.

Primer 7 *Kontekstno neodvisna gramatika za opis poljubnega programskega jezika vsebuje preveč produkcij, da bi jo lahko predstavili v celoti. Samo za občutek so na sliki 3 predstavljene produkcije, s katerimi v gramatiki programskega jezika C opišemo kontrolne stavke.*

"Simbolične majice" so tipičen primer naloge, kjer je podan vzorec, za katerega moramo ugotoviti, kateri nizi mu ustrezajo. Vzorec je podan s poenostavljenim regularnim izrazom, mali Bober pa pri reševanje take naloge gotovo ne bom imel nič več težav kot odrasel Bober.



Naloga “Opis imena datoteke” je lep primer naloge, pri kateri si lahko reševalec pomaga z znanjem regularnih izrazov. Čeprav so v nalogi uporabljeni precej enostavnejši prijemi kot v pravih regularnih izrazih (le dva enostavna operatorja), pa že samo dejstvo, da vemo za obstoj operatorjev v regularnih izrazih, pomaga pri razvozlanju in rešitvi te naloge.

Ključ za pravilno rešitev naloge je namreč v tem, da ugotovimo, da

1. zvezdica (znak `*`) predstavlja poljubno zaporedje črk,
2. vprašaj (znak `?`) pa poljubno eno črko.

Od tod naprej je rešitev enostavna:

1. Vzorec `*???as.*?` zahteva vsaj tri znake pred `a`-jem (zaradi treh vprašajev), a ime `nmas.jpg` ima le dva (namreč `nm`) in zato ne more biti rešitev naloge. Isto velja za ime `astmp.jpg`, ki pred `a`-jem nima sploh nobenega znaka.
2. Ime `nmtast.jpg` prav tako ne more biti rešitev naloge, saj vzorec `*???as.*?` zahteva piko za znakom `s`, v imenu `nmtast.jpg` pa piki sledi znak `t`.
3. Ostane še ime `mdmtexas.png`. Vzorcu `*???as.*?` se ime `mdmtexas.png` prilagodi tako, da prva zvezdica vzorca predstavlja `mdm`, trije vprašaji za prvo zvezdico predstavljajo `tex`, druga zvezdica predstavlja `pn` in zadnji vprašaj predstavlja `g`. Torej je ime `mdmtexas.png` rešitev naloge.

Osnovno vprašanje pri razlagi te naloge je naslednje: kako naj otrok ugotovi, da zvezdica in vprašaj delujeta kot operatorja (četudi tega ne poimenuje tako). Bolje je začeti z razlago regularnih izrazov, kot je opisano zgoraj, nato pa otroku prepustiti, da sam odkrije, da so v tej nalogi regularni izrazi pač nekoliko drugačni.

Kot zanimivost lahko mimogrede zapišemo, da je naloga lahko današnjim osnovnošolcem videti zapletena, čisto drugače pa bi jo sprejeli njihovi vrstniki pred dobrim desetletjem. V tistem času so namreč namesto oken in najrazličnejših brskalnikov morali uporabljati zgolj ukazno vrstico in ukaze, ki jih današnji osnovnošolci sploh ne poznajo. A če so recimo želeli prenesti skupino datotek iz ene diskete na drugo (ja, le povejmo jim, kako je bilo to takrat), so si to lahko olajšali le, če so znali uporabljati natančno tak opis datotek, kot je uporabljen v tej nalogi. Ali povedano drugače, `COMMAND.COM` je bilo praktično vse, kar so imeli na voljo, da so brkljali po disketah in diskih (ne, USB "ključkov" še ni bilo).

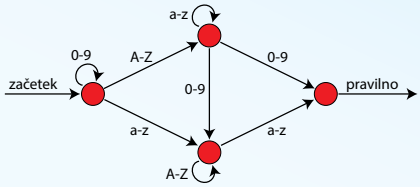
091

Preverjanje gesel

Bobrčki v neki šoli si morajo izbrati gesla za dostop do računalnikov. Da bi bilo geslo varno, mora prestati testni stroj, ki deluje takole: začnemo pri krogcu, ki ga označuje puščica »začetek«. Nato po vrsti jemljemo znake gesla (sestavljeno mora biti iz številk in črk), ki povedo, po kateri poti moramo iti z vsakega krogca. Geslo je pravilno, če končamo na krogcu, iz katerega vodi puščica »pravilno«.

Eno izmed naslednjih gesel ni pravilno? Katero?

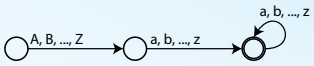
- × 123aNNa
- × Peter3ABCd
- × 2010Bober4EVEr
- × bENN0Zzz



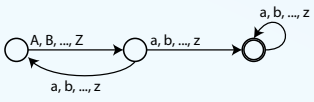
031

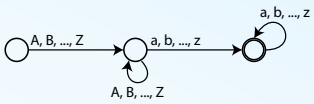
Uporabniška imena

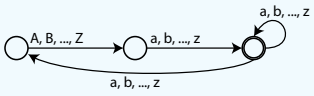
Ime se začne z veliko črko, ki mu sledi ena ali več malih črk. To opišemo takole:

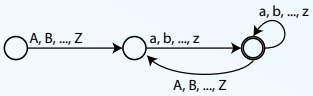


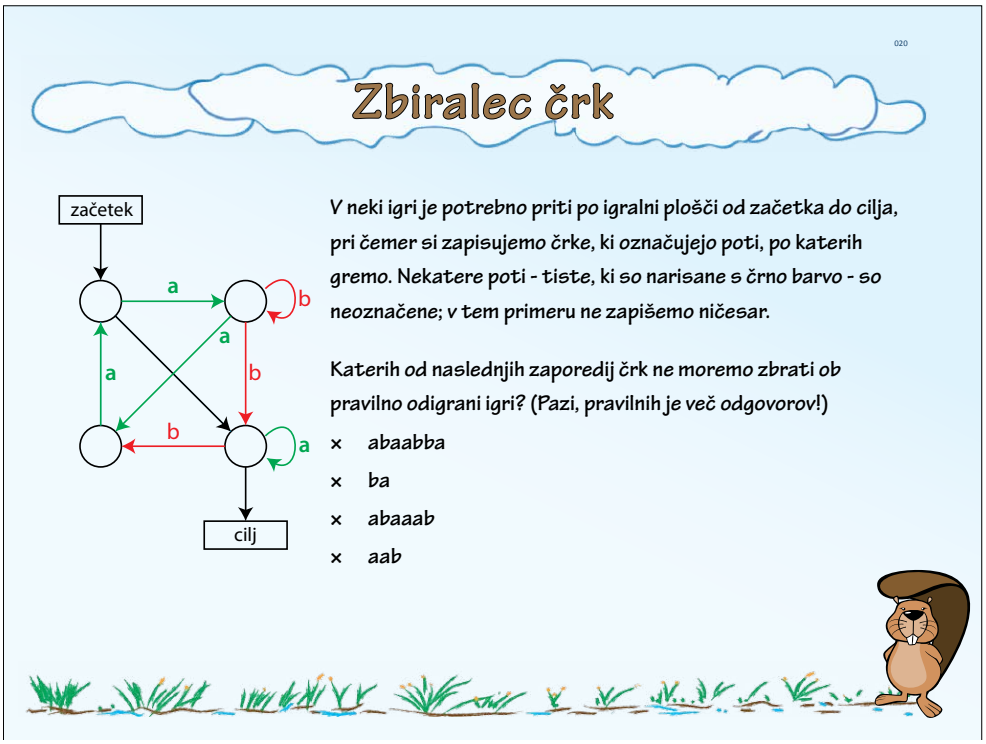
Shemo bi radi razširili tako, da bi lahko z njo opisali tudi osebe z več imeni; pisali bi jih brez presledkov, npr. FrancJožef ali MarijaTerezija. Katera od spodnjih shem je pravilna?



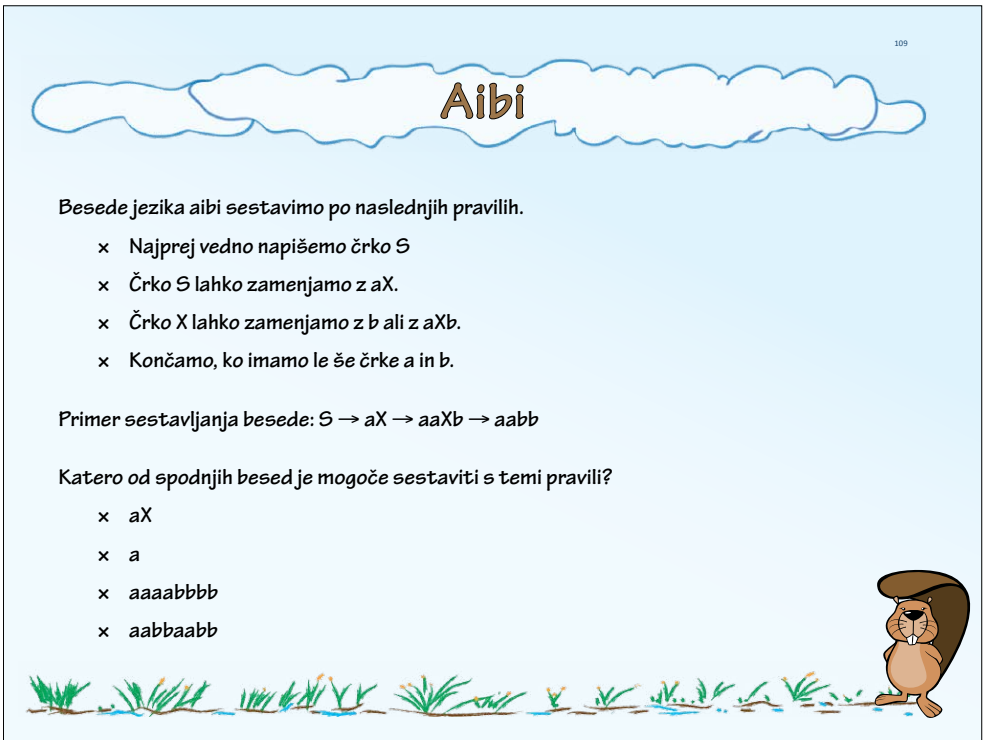








Vse tri naloge, “Preverjanje gesel”, “Uporabniška imena” in “Zbiralec črk”, so povsem klišejske naloge, ki temeljijo na ročnem izračunu delovanja končnega avtomata, kot je bilo opisano zgoraj.



"Aibi" je primer suhoparne naloge, kjer je podana kontekstno neodvisna gramatika, reševalec pa mora za podane besede poskusiti najti izpeljavo ali pa dokazati, da izpeljava ne obstaja.

Patricio Bulić

Dvojiški zapis in logika

Računalnik je elektronska naprava, ki zna računati in tako-ali-drugače obdelovati razne podatke. Celo ko v zimskih večerih v postelji, pokriti s toplo odejo, na prenosniku gledamo film, računalnik zelo intenzivno računa. Osnovni elektronski gradniki, tiste najmanjše elektronske komponente, ki jim pravimo tranzistorji, delujejo tako, da je na njihovih priključkih v katerem koli trenutku možno zaznati **le dve vrednosti električne napetosti**. No kakšen mikroelektronik in tehnolog integriranih silicijevih vezij bi temu ostro nasprotovala in trdila, da je sicer res, da tranzistorji delajo v nelinearnem območju, a so napetosti še vedno zvezne. Ampak oni živijo v svetu mikrovoltov, mikrowatov na kvadratni nanometer in podobo, kar mi niti ne opazimo in nas ne zanima – no verjetno vas res ne, mene pa vseeno "en mičkn firbec matra", a je to stvar neke druge zgodbe. Taka je pač narava teh, nam čudnih, elektronskih vezij in jo, če v današnjem času vsaj delno želimo razumeti delovanje računalnikov, moramo sprejeti. In kaj za nas, navadne uporabnike, pomeni dejstvo, da lahko na nekem priključku izredno majhnega elektronskega gradnika izmerimo le dve napetosti? To pomeni, da nam ta elektronski gradnik na tem priključku sporoča, da je v nekem trenutku nekaj "res" ali pa "ni res", ali pa da nečesa "ni" ali pa "je", mogoče celo, da je nekaj "belo" ali "črno". Torej, vsak tak, še tako majhen priključek, lahko hrani neko informacijo in se ta lahko spreminja. Največji možni količini informacije, ki jo lahko spravimo v nekaj, kar ima le dve možnosti (npr. da je nekaj "črno" ali "belo", ali da je nekaj "res" ali "ni res") pravimo ***bit***. In če se sedaj spravimo na "malo višji novo" in se oddaljimo od fizike ali elektronike, pravimo, da "**vsak osnovni gradnik računalnika lahko hrani ali obdeluje en bit informacije**". In seveda nam sedaj ne bi smelo biti tuje, če vam povem, da je tudi slika, ki jo hranite kot ozadje na zaslonu le mešanica velikega števila bitov. In da se med gledanjem vam ljubega filma, v računalniku pretaka in obdeluje ogromno število teh bitov. In, (punce pozor!) da sta Julia Roberts in Hugh Grant v filmu Nothing Hill le množica bitov, ki je v danem trenutku nekje v računalniku postavljena in določena tako, da se Julia in Hugh na zaslonu strastno poljubljata. In (pozor fantje!) da so Messi, Neymar, Iniesta in Chavi, medtem ko na prenosniku gledate njihov tik-tak, le ogromna množica bitov! In medtem ko se vi zabavate, računalnik računa z biti. Čeprav se tega ne zavedate.

Dvojiški zapis

In se vrnimo nazaj k našim bitom. Računalnikarji, medtem ko računalnike programiramo, opisujemo kdaj in kako se bodo vsi ti bitki spreminjali. Zato si radi za ti dve stanji, določamo nekakšne oznake. Lahko bi na primer zapisali kot "črno" in "belo" mogoče celo "jabolko" in "hruška". A ker smo računalnikarji potomci (v sicer izjemno kratki evoluciji) matematikov (malo tudi elektrotehnikov, ki so matematikom priskočili na pomoč s tehnično izvedbo abstraktnih postopkov), so ti določili, da bomo ti dve vrednosti zapisali kot 0 in 1. Saj nam števke pravzaprav olajšajo percepcijo računanja (čeprav bi z malo več truda in zmede lahko računali tudi z jabolkami in hruškami). V računalništvu največkrat govorimo, da ima nek ***bit vrednost 0 ali 1***.

Ker je v računalniku v enem od njegovih osnovih gradnikov mogoče hraniti in obdelovati le bite, potem mora biti vse, kar hranimo v računalniku in kar računalnik obdeluje zapisano z ničlami in enicami (ne bi pa bilo prav do fantov, ki se za nas trudijo in živijo v svetu mikrowattov, in nanometrov, da pozabimo, da sta 0 in 1 le abstrakcija dejanske napetosti na posameznem priključku). Pa si najprej za začetek poglejmo, kako zapišemo števila (kako zapisujemo oz. kodiramo informacijo vam boste lahko prebrali v drugem delu tega zvezka).

Spomnimo se še naše osnovne šole in poglejmo kako smo se učili ugotavljati vrednosti desetiških števil. Predpostavimo število 7043. Vem, da ste že "na prvi pogled" ugotovili vrednost tega števila (prvič ker vidite in ker so naši možgani izjemen procesni stroj, ki je zmožen procesirati vizuelne podatke). Ampak računalnik načeloma ne vidi (čeprav se na vsakem izpitu najde kakšen študent, ki me prepričuje v obratno). In ne zna neposredno zapisati desetiške informacije. Vrednost številu 7043 določimo takole:

7 tisočic + 0 stotic + 4 desetice + 3 enice

Če zapišemo na malo bolj formalen način, dobimo:

$$7*10^3 + 0*10^2 + 4*10^1 + 3*10^0$$

Govorimo, da je v *desetiškem* sistemu osnova *deset* (10) in vsaka števka ima lahko eno izmed *desetih* vrednosti. Pozicija oz. mesto števke v zapisu določa njeno utež. Tako ima števka na mestu 0 (skrajno desna števka) utež ena (10^0) in števka na mestu 3 (tisočica) utež 1000 (10^3). Ali bi lahko hranili desetiška števila v računalniku? **Ne neposredno**! Zakaj? Ker smo rekli, da ima posamezen priključek osnovnih gradnikov le dve možni vrednosti in ne deset! Zato moramo števila v računalniku ***zapisovati dvojiško***. V *dvojiškem sistemu* bo osnova *dve* (2), vsaka števka pa bi imela le *dve* možni vrednosti (0 in 1). In bo pozicija števke prav tako določala njeno utež (a razen enice, ne bomo mogli govoriti o deseticah, stotica, ...). Pa si poglejmo kar na primeru in skušajmo ugotoviti vrednost dvojiškega števila 1101110000011 (pa poskusite sedaj "videti" vrednost, ha!):

$$1*2^{12} + 1*2^{11} + 0*^{10} + 1*2^9 + 1*2^8 + 1*2^7 + 0*2^6 + 0*2^5 + 0*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0$$

Huh, dolga, a ne? Pa dajmo to sešteti. Pred tem si poglejmo kakšne so uteži v dvojiškem sistemu:

$2^{12} = 4096$	$2^{8} = 256$	$2^{4} = 16$	$2^{1} = 0$
$2^{11} = 2048$	$2^{7} = 128$	$2^{3} = 8$	
$2^{10} = 1024$	$2^{6} = 64$	$2^{2} = 4$	
$2^{9} = 512$	$2^{5} = 32$	$2^{1} = 2$	

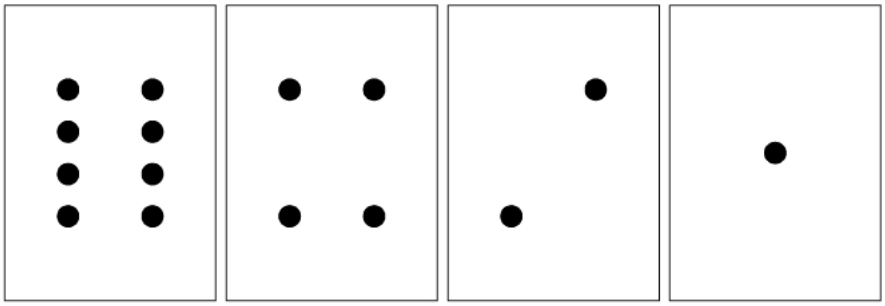
Seštejem ustrezne uteži 4096+2048+512+256+128+2+1 = 7043! No super, nismo dokazali, a mi lahko verjamete, da je v dvojiškem sistemu možno zapisati poljubno desetiško celo število, le malo več števk (=bitov) potrebujemo!

Opazimo lahko da je vsaka naslednja utež dvakratnik prejšnje. Zato lahko preprosto otrokom rečemo, da imajo za zapis števil na voljo sicer dva bita, a njihova pozicija določa vrednosti 1, 2, 4, 8, 16, 32, 128, ...

Kaj pa pretvorba iz desetiškega v dvojiško? Vidra pravi, da imajo otroci svoj način razmišljanja. Otrok ve, da ima na razpolago števila 1, 2, 4, 8, 16, 32,... Med njimi poišče največje število, ki je še manjše ali enako številu, ki ga pretvarja. Če bi npr. želeli pretvoriti število 134 v dvojiško, bi ugotovili, da je največja dvojiška utež, ki je manjša od 134 enaka 128. Zato si zapiše 1 in 128 odšteje od 134 ter dobi 6. Sedaj poskuša nprej po vrsti: najprej preveri, ali je 64 manjše od 6. Ker ni, dopiše tisti 1 eno 0 in dobi 10. Nato poskuša z 32. Ker tudi ta ni manjši od 6 dopiše še eno ničlo in dobi 100. Nato poskusi s 16. Ker tudi ta ni manjši od 6, dopiše še eno ničlo in dobi 1000. Nato poskusi z 8. Ker tudi ta ni manjši od 6, dopiše še eno ničlo in dobi 10000. Nato poskusi s 4. Ker je ta manjši od 6 dopiše eno enko in dobi 100001. Potem 4 odšteje od 6, dobi 2 ter poskuša naprej z utežjo 2. Ker je ta enaka 2 pripiše še eno 1 ter dobi 1000011. Nato odšteje 2-2 in dobi 0. Sedaj je postopek končan. Število 134, zapisano dvojiško, je torej 1000011. Tudi ostala števila (ulomljena) in neštevila (črke, barve, ...) se das zapisati z biti.

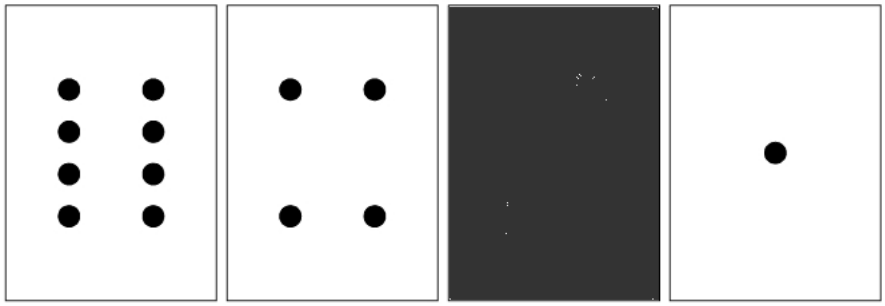
Dvojiški zapis - Bevri

Ozadje naloge je dvojiški zapis števil. Če vrednosti kovancev zapišemo od največje proti najmanjši bomo dobili zaporedje vrednosti, kot ga imajo karte z Vidre:



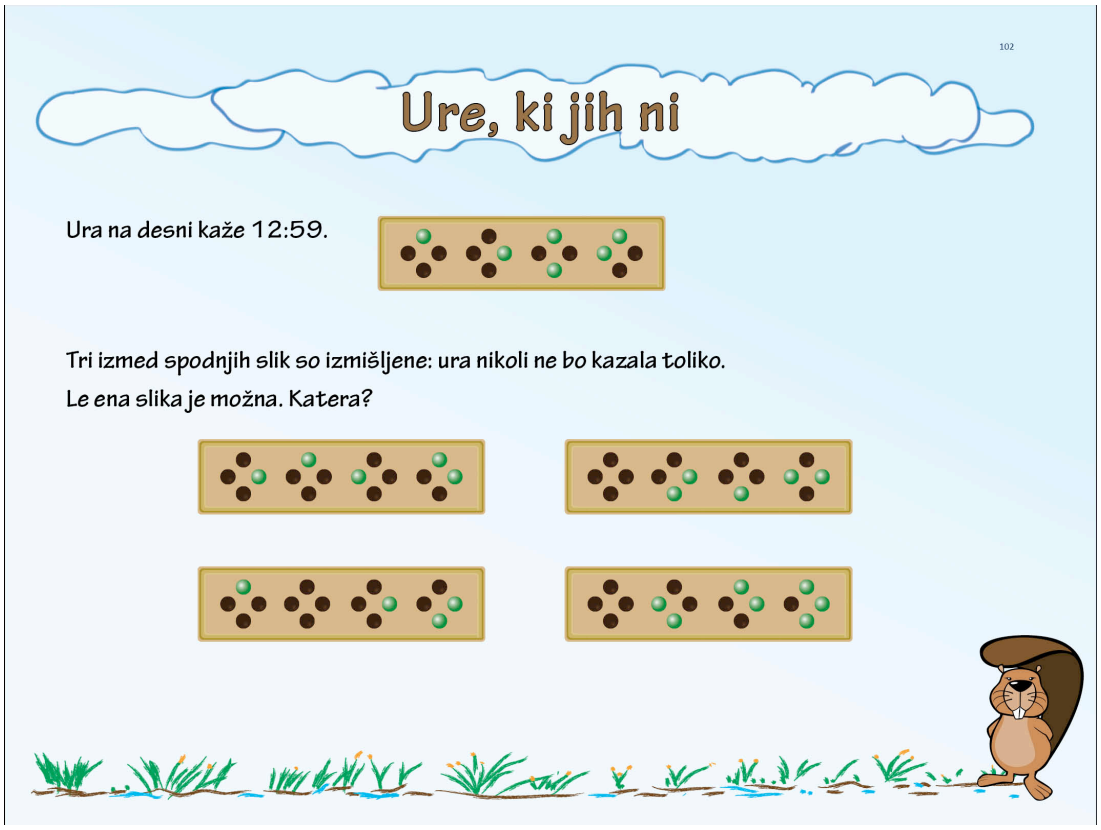
S temi kartami moramo zapisati vrednost 13. Ker karte prikazujejo vrednosti posameznega bita v 4-bitnem dvojiškem zapisu, lahko rečemo, da iščemo takšno postavitev bitov, ki nam bo dala vrednost 13. Vemo, da so vrednosti v dvojiškem zapisu določene enoznačno, torej obstaja le ena možnost zapisovanja vrednosti 13 s štirimi biti v dvojiškem zapisu. Vprašanje naloge lahko prevedemo na naslednje vprašanje: kakšno je najmanjše število bitov v 4-bitnem dvojiškem zapisu potrebnih za zapis vrednosti 13? Odgovor na to vprašanje dobimo s preprosto pretvorbo desetiške vrednosti 13 v dvojiški zapis. Spomnimo se, kaj pravi Vidra: otrok ve, da ima na razpolago števila 8, 4, 2 in 1. Med njimi bo poiskal največje število (kovanec), ki je enako ali manjše od števila, ki ga pretvarja. V našem primeru bo izbral število 8. Ker mu do 13 manjka še 5, bo izmed števil (kovancev) poiskal največje, ki je enako ali manjše od števila 5. V našem primeru bo to 4. Do 5 mu ostane le še 1, zato bo vzel kovanec z vrednostjo 1. Torej, za 13 bevrov

bo potreboval kovanec za 8 bevrov, kovanec za 4 bevre ter kovanec za 1 bevro. Če to izbiro ponazorimo z Vidrinimi kartami imamo

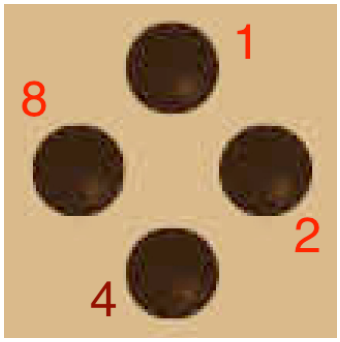


V dvojiškem zapisu je to 1101. Odgovor na vprašanje iz naloge se glasi: 3 kovance.

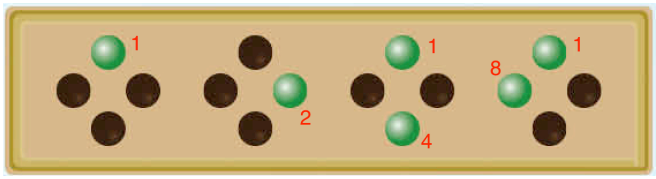
Binarni zapis – ure



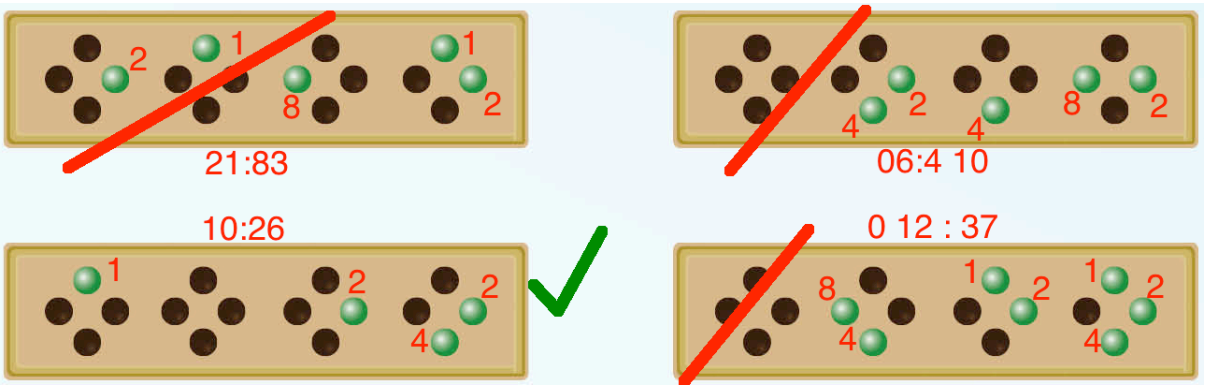
Naloga spet skriva binarni zapis števil. Vsako števko (0..9), ki tvori zapis v uri, je možno zapisati z binarnim zapisom s štirimi biti. in ravno to skriva zgornja naloga, pri čemer je vsak posamezen bit označen s črnim krogcem. Pa si poglejmo:



Z zgornjim krogcem kodiramo bit s težo 1, z desnim krogcem bit s težo 2, s spodnjim krogcem bit s težo 4 ter z levim krogcem bit s težo 8. Zeleno obarvan krogec označuje bit z vrednostjo 1 in črno obarvan krogec bit z vrednostjo 0. Tako je ura 12:59 res zapisana s spodnjo kombinacijo krogcev

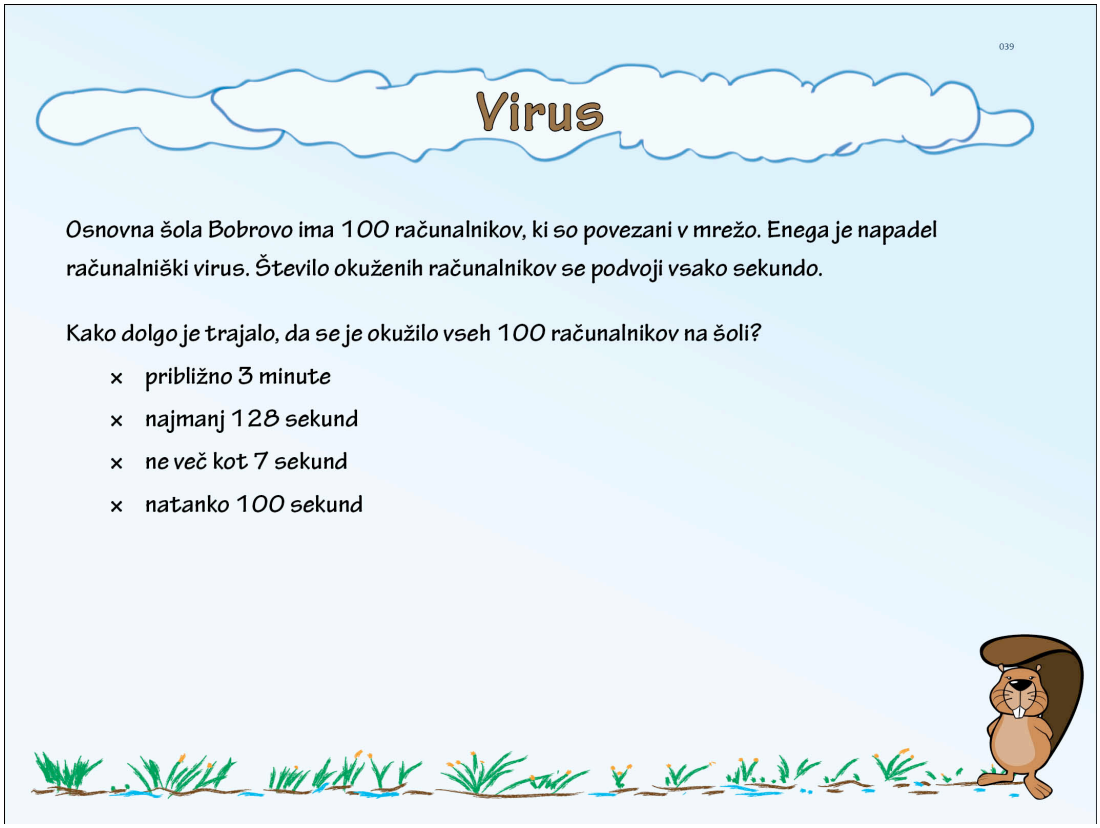


saj je v skrajno levem četvorčku krogcev z zeleno obarvan krogec na mestu bita s težo 1, tj. ti štirje krogci zapisujejo bitno kombinacijo 0001. V naslednjem četvorčku krogcev je zeleno obarvan krogec s težo 2 in je bitna kobinacija, ki jo zapisujejo ti štirje krogci 0010. V naslednjem četvorčku sta zeleno obarvana krogca s težo 1 in 4 in je pripadajoča bitna kombinacija 0101. Zadnji četvorček pa zapisuje bitno kombinacijo 1001, saj sta zeleno obarvana krogca s težo 8 in 1. Poglejmo si, kakšne vrednosti ponujajo štirje ponujeni zapisi iz naloge:



Možna je slika spodaj levo, na kateri je zapisana ura 10:26.

Virusi in eksponentna rast



Čeprav naloga omenja nekakšen hud virus, je ozadje naloge eksponentna rast. A je naloga po svoje zanimiva tudi s stališča binarnega kodiranja, saj si lahko z njim pomagamo do rešitve. Ker se število okužb vsako sekundo podvoji, nas to dejstvo močno spomni na to, da se teža vsakega bita v binarnem zapisu podvoji, če gremo iz desne proti levi. Spomnimo se, da so teže teh bitov: 1, 2, 4, 8, 16, 32, 64, 128,... Nalogo bi torej lahko prevedli na naslednje vprašanje: kako daleč se od bita s težo 1 nahaja bit s težo vsaj 100?

1 -> 2 -> 4 -> 8 -> 16 -> 32 -> 64 -> 128

Ko se okuži prvi, se bosta po eni sekundi okužila 2, po dveh sekundah 4, po treh sekundah 8, po štirih sekundah bo okuženih že 16, po petih sekundah bo okuženih 32, po šestih sekundah 64 računalnikov in po 7 sekundah pa že 128.

Nalogo bi v luči binarnega kodiranja lahko rešili tudi takole: Na kateri poziciji se nahaja bit s težo vsaj 100? spomnimo se, da se bit z najmanjšo težo (skrajno desno) nahaja na 0. mestu v binarnem zapisu, saj je njegova teža $1 = 2^0$, teža bita na 1. mestu je $2 = 2^1$, teža bita na 2. mestu je $4 = 2^2$, teža bita na 3. mestu je $8 = 2^3$, teža bita na 4. mestu je $16 = 2^4$, teža bita na 5. mestu je $32 = 2^5$, teža bita na 6. mestu je $64 = 2^6$ in teža bita na 7. mestu je $128 = 2^7$. Potrebnih je torej ne več kot 7 sekund da okužimo 128 računalnikov.

Alja lovi ravnotežje

Naloga se glasi takole:

Poglej, s čim se Alja ukvarja že celo jutro! Poredna majhna bobrovka je šla na podstrešje iskat skrite zaklade in našla staro tehtnico in škatlo s sedmimi utežmi, ki tehtajo 1, 2, 4, 8, 16, 32 in 64 kilograma. Zdaj poskuša postaviti uteži na obe strani tehtnice tako, da bo uravnotežena. Obupala je že nad tem, da bi uporabila vse uteži, in bi bila vesela že, če bi ji uspelo tehtnico uravnotežiti vsaj z nekaj utežmi.

Kateri namig bi ji lahko pomagal?

a) Uteži za 4 kg in 16 kg morata biti na nasprotnih straneh tehtnice.

b) Uteži za 4 kg in 16 kg morata biti na isti strani tehtnice.

c) Morala bi uporabiti bodisi utež za 4 kg bodisi utež za 16 kg; ti dve uteži ne moreta biti uporabljani hkrati.

d) Tehtnice ni mogoče uravnotežiti s temi utežmi.

Opazimo, da bi se dalo teže posameznih uteži zapisati z dvojiškimi števili, ki vsebujejo le eno enico. Vsako binarno število pa lahko lahko zapišemo **le na en način**. Npr. dvojiško število z vrednostjo 64 lahko zapišemo le kot 100 0000. Kaj to pomeni? Če želimo na eni strani imeti utež, ki tehta 64 kilogramov, bi morali na drugi strani nabrati uteži za 64 kilogramov. Če to obrnemo na dvojiška števila, bi z ustreznim seštevkom preostalih števil morali dobiti vrednost 64. A če seštejemo vsa preostala števila, dobimo 63, oz. dvojiško 111111. Povedano drugače: ker imamo na voljo le po en bit (1, 2, .. 64) na različnih binarnih mestih, in za zapis enega števila porabimo enaga ali več teh bitov, s preostalimi ne moremo zapisati istega števila, saj bi to pomenilo, da se eno število, da zapisati na več različnih načinov.

Otrok bi morebiti razmišljal takole: če na eno stran dam utež s težo 64 kg, mi na drugi strani ostane za 63 kg. uteži. Torej vseh uteži zagotovo ne morem uporabiti. Potem bi lahko utež za 64 kg dal stran in poskusil z utežjo za 32 kg, a bi ugotovil, da mu ne drugi strani ostane za 31 kg uteži. Tako bi s postopkom eliminacije prišel do pravilnega odgovora.

Pravilen odgovor je torej odgovor D.

Alja lovi ravnotežje 2

Podobna naloga se glasi takole:

Poglej, s čim se Alja ukvarja že celo jutro! Poredna majhna bobrovka je šla na podstrešje iskat skrite zaklade in našla staro tehtnico in škatlo s sedmimi utežmi, ki 1/8, 1/4, 1/2, 1, 2, 4 in 8 kilogramov. Zdaj poskuša postaviti uteži na obe strani tehtnice tako, da bo uravnotežena. Obupala je že nad tem, da bi uporabila vse uteži, in bi bila vesela že, če bi ji uspelo tehtnico uravnotežiti vsaj z nekaj utežmi.

Kateri namig bi ji lahko pomagal?

a) Uteži za 1/2 in 2 kg morata biti na nasprotnih straneh tehtnice.

b) Uteži za 1/2 in 2 kg morata biti na isti strani tehtnice.

c) Morala bi izpustiti bodisi utež za 1/2 kg bodisi utež za 2 kg; ti dve uteži ne moreta biti uporabljani hkrati. Enako velja za ostale pare.

d) Tehtnice ni mogoče uravnotežiti s temi utežmi.

Rešitev je enaka, kot pri zgornji nalogi. Če nam je težko razmišljati z ulomki, si pomagamo tako, da vse vrednosti uteži pomnožimo z 8.

Bobrov merski sistem

Bobrčki ne merijo dolžin v metrih. Namesto tega, uporabljajo osem merilnih palic, ki jih preprosto imenujejo A, B, C, D, E, F, G in H. Palica A je najdaljša, dolžina palice B je ena polovica dolžine palice A, dolžina palice C je ena polovica dolžine palice B in tako naprej.

Ko ljudje merimo nekaj, izražamo dolžino v metrih, na primer: 7 m in 10 cm. Bobri, ki so manj natančni, merijo dolžino s kombinacijo palic, pri čemer se lahko vsaka palica pojavlja največ enkrat. Dolžina je tako lahko, na primer A + C + D, kar preprosto zapišejo ACD. Ali recimo B + C + G, kar preprosto zapišejo kot BCG.

Bobrčki Marija, Ana, Janez in David so izmeril velikosti svojih čolničkov:

Marija: BG

Ana: BF

Janez: CEFGH

David: CDEH

Kakšen je vrstni red čolničkov, če jih razvrstimo od najdaljšega do najkrajšega?

A) David, Ana, Marija, Janez

B) Janez, David, Marija, Ana

C) Ana, Marija, David, Janez

D) Janez, David, Marija, Ana

Ozadje naloge je spet dvojiški zapis. Spomnimo se, da je utež vsakega bita pol manjša od uteži njegovega levega soseda in enkrat večja od uteži njegovega desnega soseda. Zaporedje palic ABCDEFGH lahko predstavimo dvojiško z osmimi biti. Pri tem nam npr. bit z vrednostjo 0 pomeni, da pripadajoče palice ni, bit z vrednostjo 1 pa da pripadajoča palica je.

Tako bi na primer dolžino ACD dvojiško z 8 biti zapisali kot 10110000, saj je A = 1 (torej je prisotna), B = 0, C = 1, D = 1, E = 0, F = 0, G = 0 in H = 0.

Dolžino BCG pa dvojiško zapišemo kot 01100010.

Za rešitev naloge moramo torej dvojiško zapisati vse štiri izmerjene dolžine in jih primerjati medseboj. Lotimo se najprej dvojiškega zapisa:

Marija: BG = 01000010

Ana: BF = 01000100

Janez: CEFGH = 00101111

David: CDEH = 00111001

Kako bi sedaj te vrednosti primerjali in uredili po velikosti? Poglejmo si tri možne načine.

Prvi je, da za vsako od zapisanih dvojiških števil ugotovimo desetiško vrednost. Tako je dolžina Marijinega čolna 66 (64 + 2), dolžina Aninega čolna je 68 (64 + 4), dolžina Janezovega čolna je 47 (32 + 8 + 4 + 2 + 1) ter dolžina Davidovega čolna 57 (32+16+8+1).

Izmed dveh dvojiških števil je večje tisto, ki ima najbolj levi bit pri katerem se števili razlikujeta enak 1. Poglejmo si na primeru števil 01000010 (66) in 01000100 (68). Najbolj levi bit pri katerem se števili razlikujeta je bit na mestu 2, tj. bit s težo 4: 01000**0**10 in 01000**1**00. Zato je večje drugo število. Primerjajmo na enak način še števili 00101111 in 00111001. Razlikujeta se pri bitu s težo 16 tj. 001**0**1111 < 001**1**1001. Razvrstimo sedaj na ta način binarna števila, ki kažejo velikosti čolnov:

01000100 > 01000010 > 00111001 > 00101111

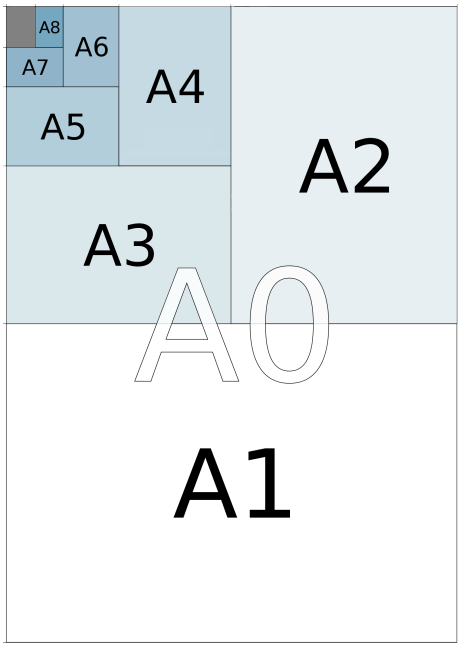
Pravkar opisan način razvrščanja pa uporabljamo pri abecednem razvrščanju! Gre za t.i. leksikografsko razvrščanje. Pri razvrščanju otrok po abecedi uporabljajo učitelji isti postopek: iščejo prvo črko v priimku (ali imenu) po kateri se dva priimka razlikujeta in na osnovi te črke določijo vrstni red. Ker naši bobrčki uporabljajo črke za merjenje dolžine in dolžina posamezne črke ustreza tudi njenemu mestu v abecedi, bi lahko izmerjene dolžine čolnov razvrstili tudi z leksikografskim razvrščanjem:

BF > BG > CDEH > CEFGH.

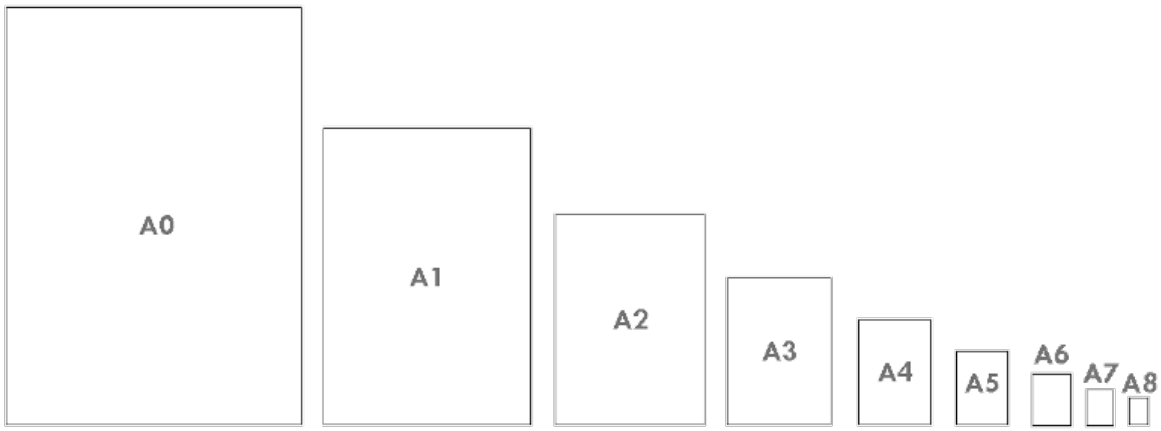
Pravilen vrstni red je Ana, Marija, David in Janez (odgovor C).

Posetnice

Standardne velikosti papirjev dobimo, kot kaže slika, iz osnovnega lista velikosti A0 (1189 x 841 mm). Če list papirja A0 razpolovimo, dobimo dva lista velikosti A1 in če sedaj razpolovimo list velikosti A1, dobimo dva lista velikost A2 ter tako naprej.



Na zalogi imamo 9 listov različnih velikosti, kot kaže spodnja slika, A0, A1, A2, A3, A4, A5, A6, A7 in A8.



Narediti želimo 19 posetnic velikosti A8 tako, da bomo porabili celotne liste papirja iz naše zaloge brez odpadkov. Katere liste naj porabimo?

A) A4, A7 in A8

B) A3 in A7s

C) A5, A6 in A8

D) A4 in A6

Ozadje naloge je spet dvojiški sistem. Če namreč vsaki velikosti papirja dodelimo ustrezno dvojiško utež (ustrezen bit v dvojiškem zapisu), potem se vprašanje iz naloge prevede na vprašanje: kateri biti morajo biti prižgani (enaki 1) v zapisu števila z vrednostjo 19? Torej velikosti A8 dodelimo težo 1 (2^0), A7 težo 2 (2^1), A6 težo 4 (2^2), A5 težo 8 (2^3), A4 težo 16 (2^4), A3 težo 32 (2^5), A2 težo 64 (2^6), A1 težo 128 (2^7) in A0 težo 256 (2^8). Z drugimi besedami, A0 list vsebuje natanko 256 listov velikosti A8, A1 list natanko 128 listov velikosti A8, A2 list natanko 64 listov A8, A3 list natanko 32 listov A8, A4 list natanko 16 listov A8, A5 list natanko 8 listov A8, A6 čist natanko 4 liste A8, A7 list natanko 2 lista A8 in list A8 natanko 1 list velikosti A8.

Število 19 v dvojiškem zapisu z devetimi biti:

$000010011 = 0*2^8 + 0*2^7 + 0*2^6 + 0*2^5 + 1*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0$

Zapisano z listi je to : 0*A0 + 0*A1 + 0*A2 + 0*A3 + 1*A4 + 0*A5 + 0*A6 + 1*A7 + 1*A8.

Pravilen odgovor je torej odgovor A).

Raznašalka pizz

Naloga spet skriva binarni zapis števil. Spomnimo se, da je vsako število v dvojiškem zapisu zapisano enoznačno, tj. unikatno. Istega števila namreč ne moremo zapisati na dva ali več načinov. Ker so bobrčki naročili 11 pizz, je to dvojiško zapisano kot 1011, tj.

$$1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 8 + 0 + 2 + 1$$

Vidimo, da so pozabili pospraviti tablo pred drugo hišo iz leve (tablo s številko 4). Če bi namreč vztrajali, da 4 ostane, ne bi nikakor mogli zapisati števila 11.

Booleova logika

Z biti moramo znati tudi računati. Osnovne operacije računanja z biti podaja Booleova algebra. Tri osnove operacijo, s katerimi je možno tvoriti poljubne funkcije znotraj Booleove algebre (ter tako implementirati vse zahteven računske postopke v računalniku), so logična vsota (ALI), logični produkt (IN) in negacija (NE). Te tri opreacije računajo izključno nad biti in jih običajno definiramo s pravilnostnimi tabelami.

Pravilnostna tabela, ki opisuje logično vsoto (ALI) je:

x	**y**	**x ALI y**
0	0	0
0	1	1
1	0	1
1	1	1

Z besedami jo opišemo takole: *Če sta vhoda x ALI y enaka 1, potem je rezultat enak 1.* Z drugimi besedami, *vsaj eden od vhodov mora biti 1*, da bo rezultat 1.

Pravilnostna tabela, ki opisuje logični produkt (IN) je:

x	**y**	**x IN y**
0	0	0
0	1	0
1	0	0
1	1	1

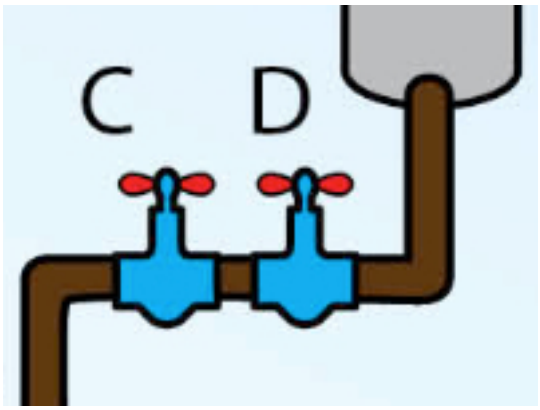
Z besedami jo opišemo takole: *Če sta vhoda x ALI y enaka 0, potem je rezultat enak 0.* Z drugimi besedami, *oba vhoda hkrati morata biti 1*, da bo rezultat 1.

Pravilnostna tabela, ki opisuje logično negacijo (NE) je:

x	**NE x**
0	1
1	0

Vodna logika

Naloga v sebi skriva Booleovo logiko oz. funkcije IN, ALI in NE. Poglejmo si najprej, kako bi prišla voda do cveta po desni cevi, tj. skozi ventila C in D (spodnja slika).



Takoj vidimo, da **morata biti odprta oba** hkrati, če želimo, da voda teče. Lahko rečemo, da ***voda mora teči skozi ventila A IN B, če želimo da priteče ven***. Pretok vode skozi ventil A in B lahko zapišemo s spodnjo pravilnostno tabelo:

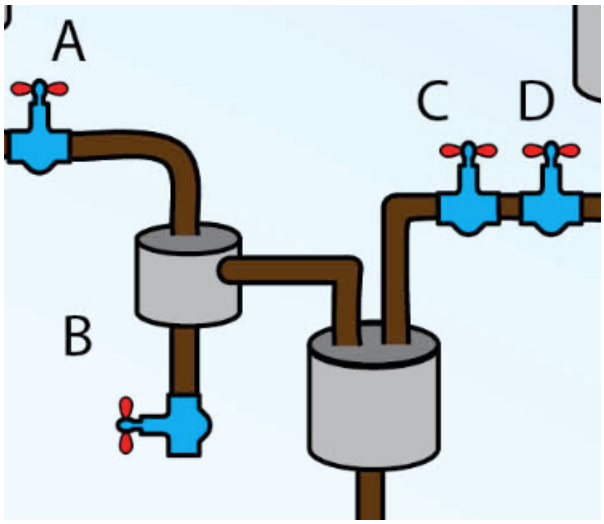
Ventil A	Ventil B	Izhod
Ne teče	Ne teče	Ne teče
Ne teče	Teče	Ne teče
Teče	Ne teče	Ne teče
Teče	Teče	Teče

Če bi pojav, da voda teče, "kodirali" z 1, in da voda ne teče "kodirali " z 0, bi lahko v Booleovi algebri to opisali s pravilnostno tabelo:

Ventil A	Ventil B	Izhod
0	0	0
0	1	0
1	0	0
1	1	1

Ta tabela pa opisuje logično operacijo IN.

Po levi cevi voda priteče le, če je odprt ventil A. Kaj pa skozi osrednjo posodo do cveta (spodnja slika)?



Če odmislimo ventil B, ki mora biti vedno zaprt, bo ***voda skozi osrednjo posodo tekla, ko bo odprt ventil A ALI, ko bosta odprta oba ventila C in D.*** Pretok vode skozi osrednjo posodo lahko zapišemo s spodnjo pravilnostno tabelo:

Ventil A	**Ventila B in C**	**Izhod**
Ne teče	Ne teče	Ne teče
Ne teče	Teče	Teče
Teče	Ne teče	Teče
Teče	Teče	Teče

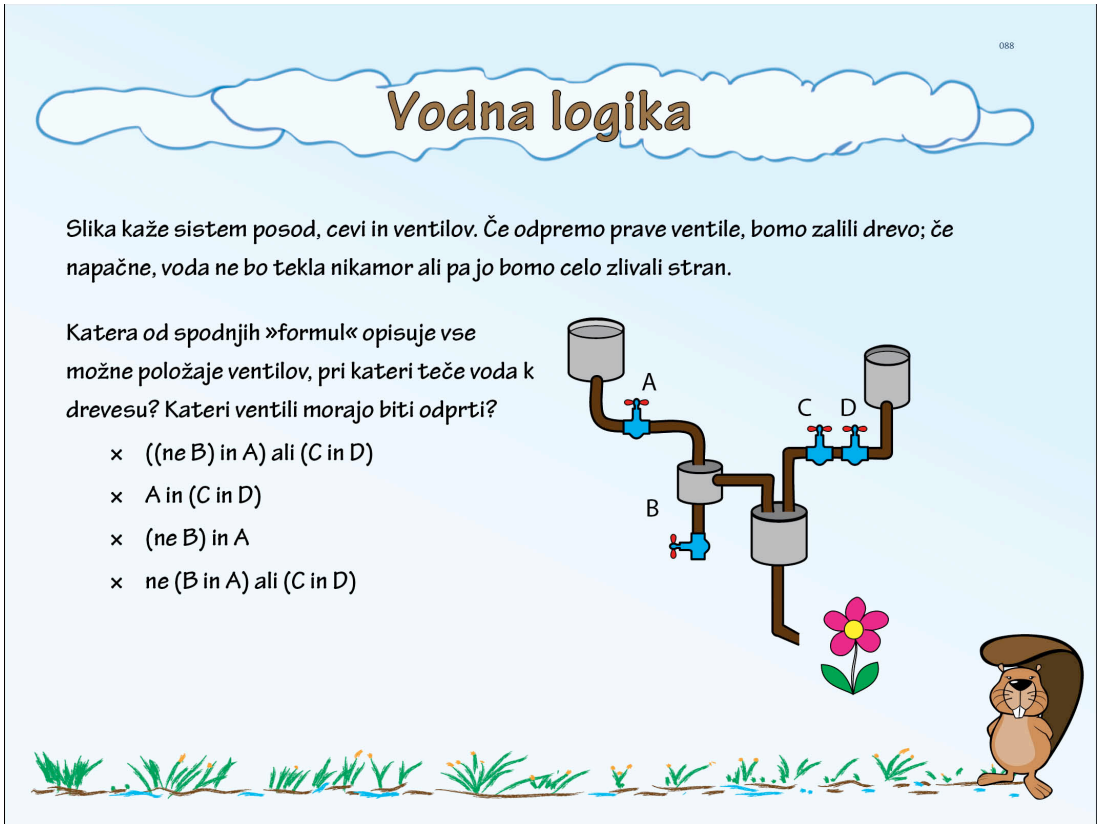
Če bi pojav, da voda teče, "kodirali" z 1, in da voda ne teče "kodirali " z 0, bi lahko v Booleovi algebri to opisali s pravilnostno tabelo:

Ventil A	**Ventil B**	**Izhod**
0	0	0
0	1	1
1	0	1
1	1	1

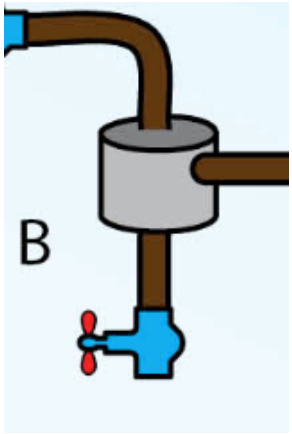
Ta tabela pa opisuje logično operacijo ALI.

Naloga je sila preprosta in jo otroci rešijo brez težav brez poznavanja Booleove logike. Ker je v vseh ponujenih odgovorih ventil C vedno zaprt, voda ne teže skozi desno cev. Edina možnost da zalijemo cvet je, da je B zaprt in A odprt (prvi odgovor).

Vodna logika 2



Naloga v sebi spet skriva Booleovo logiko oz. operacije IN, ALI in NE. In in ALI smo predstavili v prejšnji nalogi. Kje se skriva NE? Poglejmo si pomen postavitve posode z odtokom pred ventilom B:



Voda bo iz posode odtekala po desni cevi, če bo ventil zaprt, sicer bo šla skozi ventil.

Zapišimo to s pravilnostno tabelo:

Ventil B	**Izhod (desna cev)**
Ne teče	Teče
Teče	Ne teče

Če bi pojav, da voda teče, "kodirali" z 1, in da voda ne teče "kodirali " z 0, bi lahko v Booleovi algebri to opisali s pravilnostno tabelo:

Ventil B	**Izhod (desna cev)**
0	1
1	0

Ta tabela opisuje logično operacijo NE.

Če sedaj z logičnimi operacijami zapišemo "formulo" za zalivanje, je ta:

((NE B) IN A) ALI (C IN D)

Uroš Lotrič

Kodi

Kodiranje je pomembno pri stiskanju podatkov, popravljanju napak pri prenosu in šifriranju. Algoritmi za stiskanje podatkov poskušajo podatke predstaviti tako, da ohranijo vsebino in pri tem uporabijo občutno manj prostora. Pri prenosu podatkov se napakam zaradi motenj ne moremo izogniti, na srečo pa se lahko pred njimi dobro zavarujemo. S šifriranjem podatkov želimo preprečiti, da jih na poti od pošiljatelja do prejemnika ne bi prestregli vsiljivci in se z njimi okoristili.

V večini današnjih računalnikov so podatki predstavljeni z nizi, sestavljenimi iz dveh simbolov. Simbola sta lahko leva in desna stran, bela in črna kroglica, ničla in enica. Na primer, slovenska abeceda ima 25 različnih črk. Ena od možnosti je, da vsako črko predstavimo z nizom dolžine pet, v katerem nastopata samo dva različna simbola. Iz nizov dolžine pet lahko zgradimo 2^5=32 različnih vzorcev, kar je dovolj za 25 črk slovenske abecede in še za nekaj posebnih znakov. Lahko bi si zamislili, da znaku A priredimo niz 00001, znaku B niz 00010, … Takemu postopku prirejanja nizov pravimo kodiranje, preslikavi pa kod. Kod preslika vsak znak osnovne abecede v kodno zamenjavo, ki je sestavljena iz znakov kodne abecede.

Nekateri kodi so zelo razširjeni. Eden bolj uporabljanih kodov v računalništvu je kod ASCII, ki na podoben način kot zgoraj osnovnim znakom prireja nize dolžine sedem:

0 → 00110001, 1 → 00110010, …
A → 01000001, B → 01000010, …
a → 11000001, b → 11000010

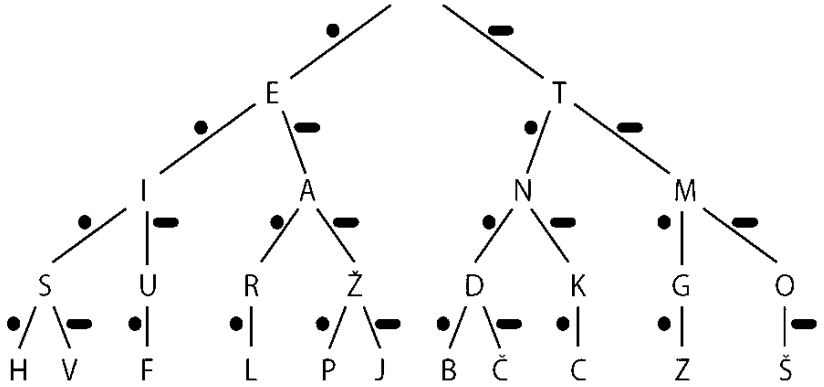
Kodi so seveda lahko tudi precej drugačni. Znaki so lahko kodirani tudi z različno dolgimi nizi ali kodnimi zamenjavami:

A → 0, B → 0 1, C → 0 1 0 .

Za potrebe telegrafiranja so sredi 19. stoletja razvili Morsejev kod. Sestavljen je iz kratkih in dolgih piskov ter pavz. Na papir se te znake zapisuje kot piko, črto in presledek. Najbolj pogosta črka angleške abecede E ima za kodno zamenjavo piko, malenkost manj pogosta črka A pa piko in črto. Kod za vsako črko angleške abecede in za vsako cifro uporablja edinstveno kodno zamenjavo.

Vsakodnevno se srečujemo še s črtno kodo, ki je ravno tako kod, pri katerem kodno abecedo sestavljata tanka in debela črta.

Kod je lahko kakršna koli preslikava. Najlepše ga predstavimo s kodnim drevesom. Vozlišča v drevesu označujejo znake, pot od korena do izbranega vozlišča pa nam določi kodno zamenjavo. Na primer, pri sprehodu po Morsejevem kodnem drevesu od korena do znaka C sestavimo kodno zamenjavo črta, pika, črta, pika.



Seveda vse preslikave niso smiselne in med smiselnimi tudi niso vse enako dobre. Kod ni singularen, če ima vsak znak svojo kodno zamenjavo. Pri takem kodu lahko iz kodnih zamenjav rekonstruiramo osnovne znake. Pri singularnem kodu to ne gre. Primer:

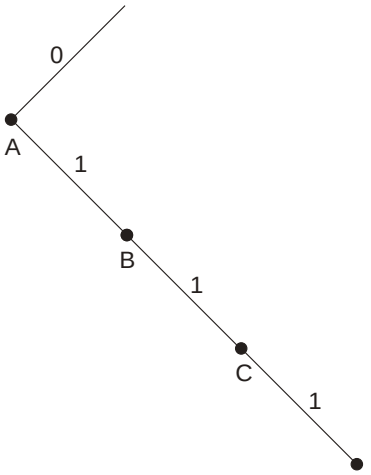
Znak	Singularni kod	Nesingularni kod
A	00	0
B	10	10
C	01	00
D	10	01

Zgornji nesingualrni kod še vedno ni najboljši, saj kodni zapis 00 lahko razumemo kot AA ali pa kot C. Podobno težavo srečamo tudi pri Morsejev kodu: ena črta predstavlja T, dve črti predstavljata M. Morsejev kod se je v zgodovini vseeno veliko uporabljal. Kako so se izognili težavi?

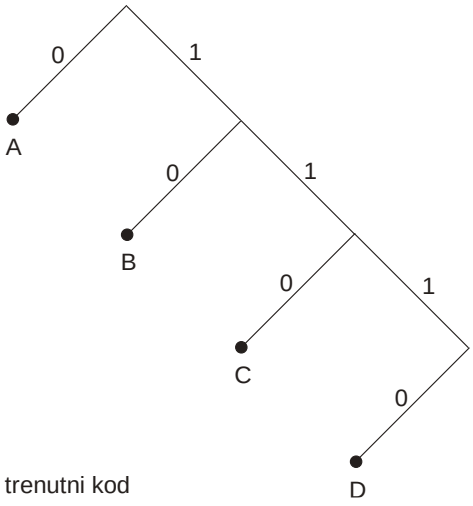
Uporabni so enoznačni kodi, za katere velja, da jih lahko rekonstruiramo na en sam način. Zaradi čim hitrejše obdelave podatkov se največ uporabljajo trenutni kodi. To so kodi, pri katerih lahko osnovni znak rekonstruiramo takoj ko smo prejeli zadnji znak kodne zamenjave. Spodnja tabela kaže dva podobna koda.

Znak	Enoznačni	Trenutni
A	0	0
B	01	10
C	011	110
D	0111	1110

Prvi je samo enoznačni, saj znak lahko rekonstruiramo iz števila zaporednih enic šele potem, ko prejmemo prvi znak (0) kodne zamenjave naslednjega znaka. Veliko boljši je trenutni kod, saj je 0 vedno zadnji znak kodne zamenjave – ko jo opazimo, preštejemo enke pred njo in rekonstruiramo znak. Naslednja slika prikazuje enoznačni in trenutni kod iz zgornje tabele. Trenutni kodi se od ostalih razlikujejo po tem, da imajo kodne zamenjave samo na listih, ne pa tudi na vmesnih vozliščih.



enoznačni kod



trenutni kod

Naloge, v katerih se skrivajo osnovne ideje kodiranja, vključujejo zamenjevanja znakov osnovne abecede v znake kodirne abecede, enoznačnost kodov in dopolnjevanje kodne preslikave.

053

Kodiranje

Bobra Barbara in Benjamin sta si izmislila nenavaden način zapisovanja besedil s številkami. Sestavila sta tabelo na desni. Vsaki črki ustreza številka; ko želita zapisati določeno črko, zapišeta dvakratnik ustrezne številke. Črko B zapišeta s številko 4; črko M zapišeta z 38. Besedo BOBER bi zapisala kot 45241256.

A	1	B	2	C	3	Č	4	D	5
E	6	F	7	G	8	H	9	I	15
J	16	K	17	L	18	M	19	N	25
O	26	P	27	R	28	S	29	Š	35
T	36	U	37	V	38	Z	39	Ž	45

Katero besedo predstavlja številka 562874502503212?

Številke stanovanj

Bober Peter opazuje sosednji kompleks mravljišč. Sestavljen je iz desetih mravljišč. Vsako ima pet vhodov. Za vsakim vhodom je 12 nadstropij. Prvo nadstropje ima 120 prostorov, ostala pa 80.

Mravlje so prostore oštevilčile tako, da prva številka označuje številko mravljišča (1-10), druga številko vhoda, tretja nadstropje in četrta prostor v tem nadstropju. Tako ima prostor 2 nadstropja 3 vhoda 4 mravljišča 8 številko 8432.

Petru se zdi to oštevilčenje nespametno. Pravi, da se bo pogosto dogajalo, da bodo imeli različni prostori isto številko. In res, od spodnjih štirih številk je kar tri možno brati na različne načine. Le za eno jasno, kateri prostor predstavlja. Za katero?

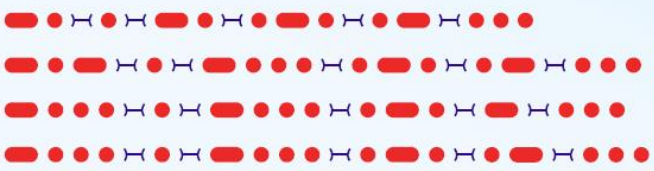
11122 14111 12131 11113

Piskajoči bobri

Pasma piskajočih bobrov si pošilja sporočila s kratkimi in dolgimi piski (zapisovali jih bomo s pikami in vejicami). Tule je nekaj črk:

Med črkami delajo kratke presledke, ki jih bomo zapisali z znakom

Eno od spodnjih zaporedij predstavlja besedo BEBRAS. Katero?



Stiskanje podatkov

Stiskanje je postopek, ki zmanjša velikost datotek, ne da bi resno zmanjšali integriteto podatkov. Stiskanje je na veliko uporabljano v moderni informacijski tehnologiji za zmanjšane tekstovnih, glasbenih, slikovnih datotek in video vsebin. Obstaja množica tehnik za stiskanje.

V grobem ločimo stiskanje brez izgub in stiskanje z izgubami. Medtem ko je pri prvem pomembno, da lahko stisnjene podatke raztegnemo nazaj v prvotno obliko, pri drugem dovolimo, da nekaj vsebine izgubimo. Slednji postopki se uporabljajo veliko pri stiskanju slik ter zvočnih in video zapisov. Izkoriščajo dejstvo, da ljudje zaradi fizične omejitve čutil ne zaznamo vseh podrobnosti, ki jih zabeležijo tipala v digitalnih napravah.

Vzemimo, da govorimo abecedo, ki pozna presledek in tri črke {_, A, B, C}. Kod, ki bi ga lahko uporabili je _ - 00, A – 01, B – 10, C – 11. Daljši niz v naši abecedi bi potem na primer kodirali takole:

A_CAA_ABA_BCAA_A → 01001101010001100100101101010001

Pa znamo bolje?

Huffmanov kod

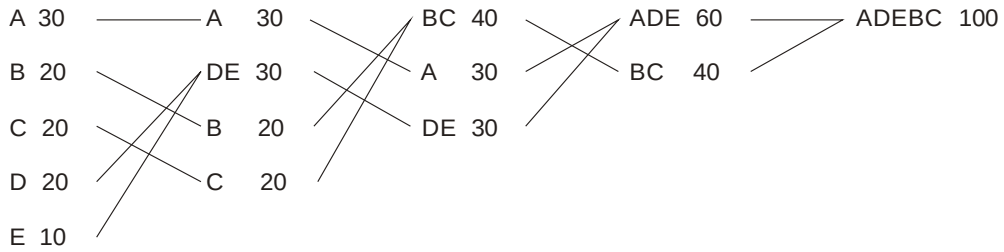
Osnovna ideja pri stiskanju je, da bolj pogoste znake zapišemo s krajšimi kodnimi zamenjavami, manj pogoste pa z daljšimi kodnimi zamenjavami. To idejo najbolje udejanja Huffmanov kod. V nadaljevanju si bomo ogledali, kako se z njim dobi kodne zamenjave, ki uporabljajo binarno kodno abecedo.

Vzemimo preprost primer, ko želimo kodirati dva znaka {A, B}. Prvemu bomo dodelili kodno zamenjavo 0, drugemu kodno zamenjavo 1.

Bolj zapleteno je, če želimo kodirati tri znake {A, B, C}. Predpostavimo, da je znak A najbolj pogost znak v našem zapisu. Huffman je ugotovil, da je najbolj smiselno najmanj pogosta znaka (B in C) združiti v nov znak BC. Zdaj, ko imamo samo dva znaka, jima lahko dodelimo kodni zamenjavi tako kot v prejšnjem primeru: A nadomestimo z 0, BC pa z 1. Sestavljeni znak BC razdružimo tako, da k enici, dodeljeni sestavljenemu znaku, enkrat pripišemo ničlo, drugič pa enico. Dobimo končne kodne zamenjave: A: 0, B: 10, C: 11. Huffmanov kod ni enolično določen – enako dobro bi bilo, če bi znakom dodelili naslednje kodne zamenjave: A: 0, B: 11, C: 10 ali pa A: 1, B: 01, C: 00.

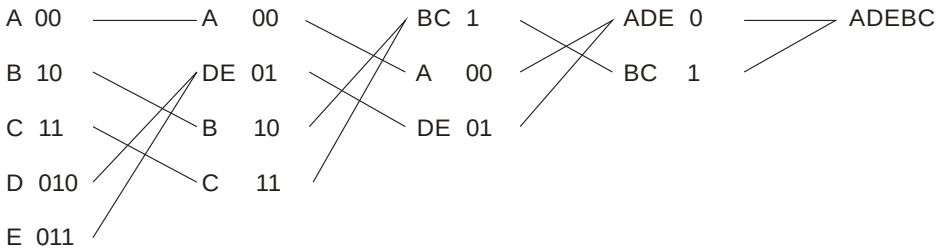
Posplošimo. Huffmanov postopek kodiranja poteka v dveh fazah. V prvi fazi pare najmanj pogostih znakov združujemo v sestavljene znake. Postopek ponavljamo, dokler nam ne ostane en sam sestavljeni znak. V drugi fazi sestavljene znake razdružujemo, pri čemer enemu (sestavljenemu) znaku dodelimo 0, drugemu pa 1.

Poglejmo primer. Imamo pet znakov {A, B, C, D, E}. Število posameznih znakov v zapisu je {30, 20, 20, 20, 10}. Postopek kodiranja gradnje Huffmanovega koda je grafično prikazan na spodnji sliki.

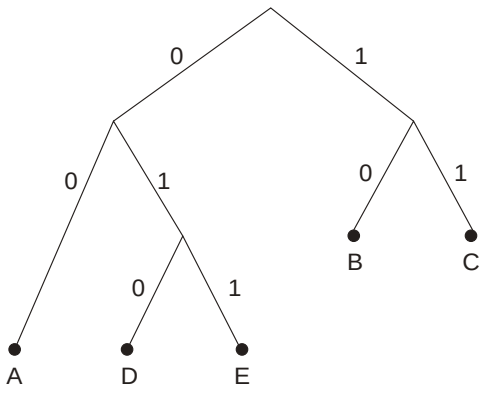


Najprej vzamemo najmanj pogosta znaka, to sta gotovo E in katerikoli od znakov B, C, D. Če izberemo D, nam ostanejo štirje znaki {A, DE, B, C } s pogostostmi {30, 30, 20, 20}. Šteli smo, da se znak DE pojavi, če se pojavi znak D ali znak E. Postopek nadaljujemo na enak način. Spet združimo najmanj pogosta znaka, to sta B in C. Dobimo znake {BC, A, DE} in pogostosti pojavitve {40, 30, 30}. Postopek ponovimo še dvakrat. Najprej dobimo {ADE, BC}, pri čemer se znaki A, D ali E pojavijo 60-krat, znaka B ali C pa 40-krat. Na koncu oba znaka združimo še v en sam znak.

V drugi fazi vsak sestavljen znak razdelimo na znaka, iz katerih smo ga v prvi fazi sestavili. Enemu od teh dveh znakov dodelimo znak kodne abecede 0, drugemu pa 1. Vzemimo, da znaku na zgornji veji dodelimo 0, znaku na spodnji veji pa 1. Dodelitev kodnih zamenjav je prikazana na spodnji sliki.



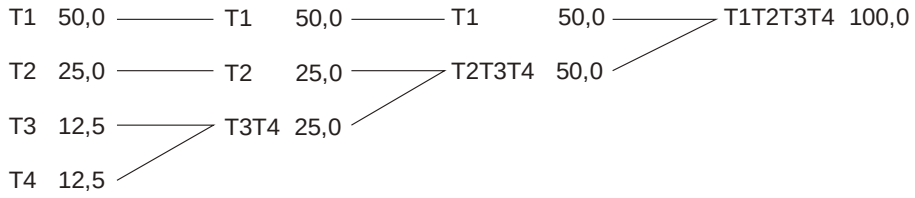
Znak ADEBC sestavljata znaka ADE in BC. Znak ADE je narisan zgoraj, zato mu dodelimo 0, znaku BC pa 1. V naslednjem koraku se znak BC ne razcepi, zato se njegova kodna zamenjava ne spremeni. Znak ADE se razcepi na A in DE, zato 0 iz prejšnjega koraka pri A dopišemo 0, pri DE pa 1. Če na enak način nadaljujemo do konca, dobimo kodne zamenjave, ki jih bolj elegantno predstavimo s kodnim drevesom.

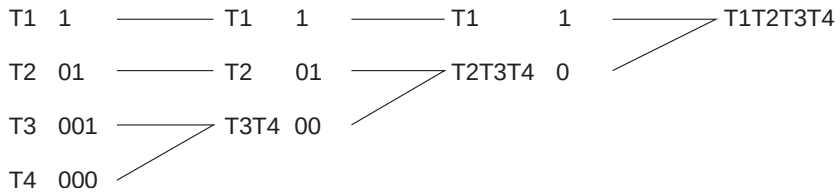


Najbolj verjetni znak A kodiramo samo z dvema znakoma kodne abecede, najmanj verjetni znak E pa s tremi.

Kodne zamenjave si lahko predstavljamo tudi kot odgovore na vprašanja, ki nas od korena pripeljejo do želene črke. Vprašanja morajo biti zastavljena tako, da na njih lahko odgovorimo z ne (0) ali z da (1).

Poglejmo primer. Na tekmi so nastopili štirje tekmovalci {T1, T2, T3, T4}. Verjetnost za zmago prvega je bila 50 %, drugega 25 %, tretjega in četrtega pa 12,5 %. Tekme nismo spremljali, radi pa bi s čim manj vprašanji izvedeli, kdo je zmagal. Zato najprej zgradimo Huffmanovo drevo. Ali lahko z verjetnostjo delamo tako, kot smo prej s pogostostjo pojavitve?

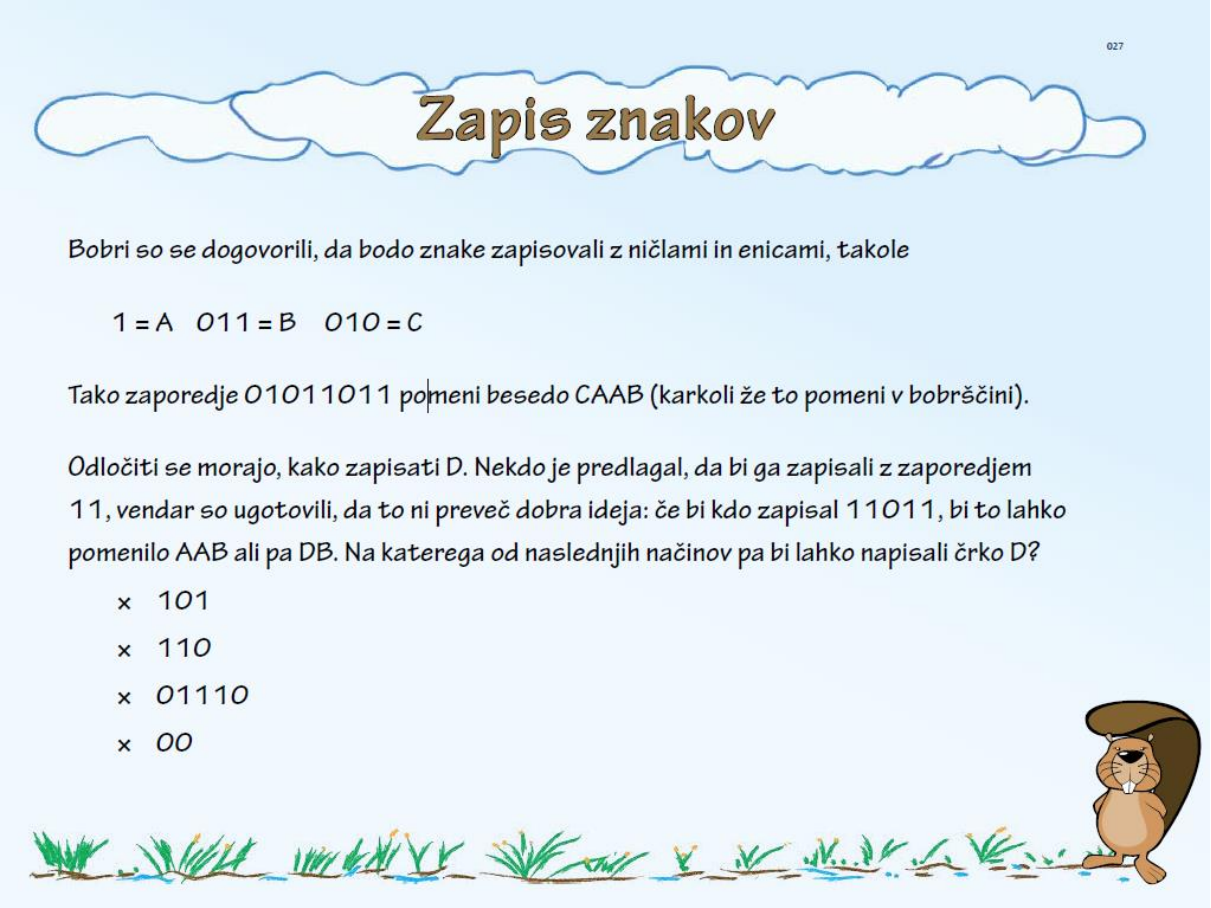




Na podlagi drevesa poskusimo oblikovati vprašanja. Tekmovalca T1 od ostalih ločimo z vprašanjem »Ali je zmagal tekmovalec T1?« To je tudi najbolj smiselno vprašanje, saj je zelo verjetno, da je zmagal. Drugo vprašanje bi bilo »Ali je zmagal tekmovalec T2?«, saj je verjetnost za njegovo zmago bistveno večja od verjetnosti za zmago tekmovalca T3 ali tekmovalca T4. Če ni zmagal niti tekmovalec T1 niti tekmovalec T2, z vprašanjem »Ali je zmagal tekmovalec T3?« dokončno razblinimo dvome.

Opisani postopek gradnje kodnih zamenjav uporablja večina modernih algoritmov za stiskanje podatkov (ZIP, RAR, JPG, …) . Največji problem pri stiskanju podatkov v praksi je, da ne poznamo verjetnosti. Različni postopki za stiskanje se tako razlikujejo predvsem v načinu ocenjevanja verjetnosti.

Poznavanje trenutnih kodov in risanje kodnih dreves se skriva za nalogama Zapis znakov in Bobrovsko kodiranje.



BOBROVSKO KODIRANJE

Bobri prevažajo hlode po reki s splavom. Hlodi so po velikosti lahko majhni (M), srednje velki (S), veliki (V) ali zelo veliki (Z). Da označijo, kakšni hlodi so na splavu, uporabljajo poseben kodirni sistem. Kadar splav pride do jezu, s pomočjo opozorilnega roga in posebnih kod opišejo hlode na splavu, recimo MZZZMVVS. Na opozorilnem rogu znajo zapiskati nizek (○) in visok ton (◊). Med vsakim piskom je sekundni odmor. Bober Brane si je zamislil štiri kodirne tabele za velikosti hlodov, ki so opisane spodaj. Žal je zgolj ena sprejemljiva. Katera?

A) M: ◊	S: ◊◊	V: ◊◊◊	Z: ◊◊◊◊
B) M: ◊	S: ○	V: ◊◊	Z: ○◊
C) M: ◊	S: ○◊	V: ○○◊	Z: ○○○
D) M: ○	S: ○◊	V: ○○	Z: ○◊○

S Huffmanovim kodom si lahko pomagamo tudi pri nalogi kot je Ugibanje števila.

111

Ugibanje števila

Bobri so radi v šoli, le med odmori jim je dolgčas. Zato se pogosto igrajo ugibanje števil: eden si zamisli število med 1 in 100, drugi ga poskuša ugibati. Ta, ki si je zamislil število, ob vsakem ugibanju pove, ali je iskano število večje ali manjše.

Bobrovka Hana vedno ugiba tako, da začne pri številu 50. Če je iskano število manjše, bo nadaljevala s 25, če večje s 75. Tako nadaljuje: v vsakem koraku pove število, ki je na sredi med najmanjšim in največjim kandidatom.

Kolikokrat, največ, mora ugibati, preden ugane?

- × 7
- × 16
- × 40
- × 50

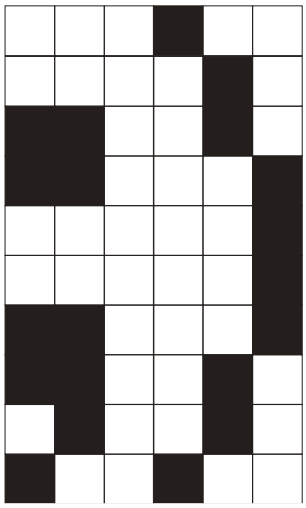
Verižni kodi

Kadar želimo kodirati znak za znakom, dobimo najboljše rešitve z gradnjo Huffmanovega koda. Če za to ni potrebe, lahko stiskamo še bolj temeljito. Ena od takih tehnik je verižno kodiranje.

Ideja verižnega kodiranja ali kodiranja z dolžinami nizov je zelo preprosta. Izkorišča dejstvo, da se v zapisih vzorci ponavljajo. Namesto večkratnega zapisovanja enakega vzorca verižno kodiranje zapiše en vzorec in število ponovitev. Na primer, niz znakov AAAABBC bi z verižnim kodiranjem zapisali kot 4A2B1C. Težava se pojavi, ko se znaki ne ponavljajo. Takrat z verižnem kodiranjem dobimo nasprotni učinek – zapis se podaljša. Na primer niz ABCBAC bi zapisali kot 1A1B1C1B1A1C in namesto šestih uporabili kar dvanajst znakov! Običajno se verižno kodiranje zato uporablja samo, kadar z njim skrajšamo zapis, drugače se kodira znak po znak. Niz znakov AAAABBC, verižno kodiran z omenjeno dopolnitvijo, je potem 4ABBC. Za enega ali dva enaka zaporedna znaka kodiranje z verižnim kodom ni smiselno, zato jih ohranimo v osnovnem zapisu.

Verižno kodiranje srečamo v napravah za faksiranje sporočil (standard ITU-T4), v slikovnih formatih BMP, TIFF, PCX.

Sliko, na primer, si lahko predstavljamo kot zaporedje točk, ki si sledijo vrstico za vrstico. Omejimo se na črno bele slike, na primer na tako, kot je spodnja.



Čisto osnovno verižno kodiranje nas pripelje do opisa

B3 Č1 B6 Č1 B1 Č2 B2 Č1 B1 Č2 B3 Č1 B5 Č1 B5 Č3 B3 Č3 B2 Č1 B2 Č1 B2 Č1 B1 Č1 B2 Č1 B2 .

Pri tem zapisu je prva težava ta, da moramo v naprej vedeti, kako široka je slika. Temu se lahko ognemo. Običajno kodiramo vsako vrstico posebej:

B3 Č1 B2 B4 Č1 B1 Č2 B2 Č1 B1 Č2 B3 Č1 B5 Č1 B5 Č1 Č2 B3 Č1 Č2 B2 Č1 B1
B1 Č1 B2 Č1 B1 Č1 B2 Č1 B2 .

V zgornjem zapisu so samo zaradi jasnosti med opise posameznih vrstic dodani večji presledki.

Za črno-bele slike lahko ta zapis v marsičem poenostavimo. Ni nam potrebno sporočati, koliko belih pik sledi zadnji črni, lahko samo povemo, da se je vrstica končala. Nadalje lahko upoštevamo, da se črna in bela barva izmenjujeta – beli sledi vedno črna in obratno. Edino, česat ne vemo, je, kakšne barve je prva točka v vrstici. Ena možnost je, da opis vrstice vedno začnemo z belo barvo, druga pa, da barvo vsakič pošljemo. Če uporabimo prvo idejo, bi bil zapis enak

3 1 | 4 1 | 0 2 2 1 | 0 2 3 1| 5 1 | 5 1 | 0 2 3 1 | 0 2 2 1 | 1 1 2 1 | 0 1 2 1 | ,

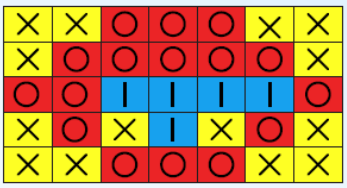
kjer navpičnica predstavlja znak za konec vrstice.

To idejo kodiranja pri pošiljanju podatkov uporabljajo faksi. V praksi je stvar še malo bolj zapletena. Za učinkovito pošiljanje so dolžine belih in črnih polj ter konce vrstic kodirali še z dvema Huffmanovima kodoma – kodom za bele in kodom za črne pike. Drevesa so zgradili po analizi pogostosti posameznih dolžin v množici tipičnih dokumentov.

Pri sestavljanju nalog Zapisovanje slike in Slika iz pik so bile uporabljene ravno zgornje ideje.

Zapisovanje slike

Kako bi spremenili sliko v zaporedje znakov? Eden od načinov je takšen:



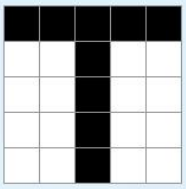
bxcobx
axeoax
???
axaoaxaiaxaoax
bxcobx

Vsako vrstico slike opisuje zaporedje na desni. Kako pa bi opisali manjkajočo srednjo vrstico?

Slika iz pik

Sliko, sestavljeno iz pobarvanih ali praznih kvadratkov na mreži, lahko opišemo s številkami, kot kaže slika.

0, 5
2, 1, 2
2, 1, 2
2, 1, 2
2, 1, 2



V vsaki vrstici se izmenjujejo beli in črni kvadratki. Prva številka pove, s koliko belimi kvadratki se začne slika, druga številka pove, koliko črnih jim sledi, tretja spet pove število belih kvadratkov, druga črnih in tako naprej, dokler je potrebno.

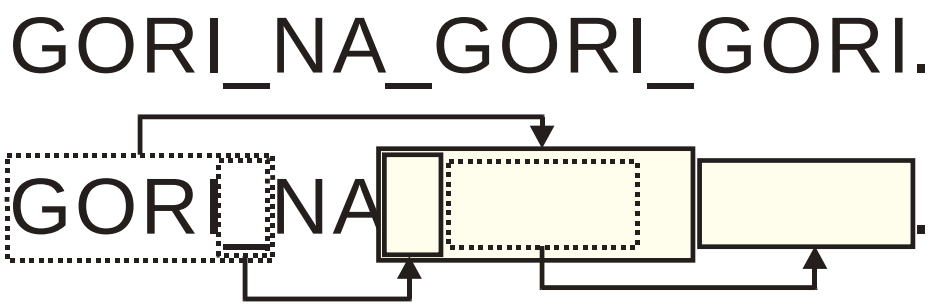
Katera črka bi se pokazala, če bi izrisali tole sliko?

0,1,3,1
0,1,3,1
0,5
0,1,3,1
0,1,3,1

Stiskanje z iskanjem podobnih vzorcev

Ta način stiskanja uporabljajo programi za stiskanje datotek, na primer ZIP. Ideja za to stiskanje prihaja iz analize jezikov, v katerih zaporedja črk niso popolnoma naključna, ampak se vzorci od časa do časa ponavljajo. Stiskanje je učinkovito, če si zapomnimo nize znakov, ki so se že pojavili. Ko najdemo enak niz znakov, nato elegantno sporočimo samo koliko znakov nazaj se je niz že pojavil in kako dolg je bil.

Poglejmo primer:



V tekstovni obliki lahko zapišemo v obliki

GORI_NA(3,1)(8,5)(5,4).

Pari številk v oklepajih predstavljajo sklic na obstoječi niz. Prva številka označuje, koliko znakov nazaj se niz začne, druga pa njegovo dolžino.

Obstoječi nizi se prepisujejo znak po znak, zato si lahko privoščimo tudi sklic na niz, ki je daljši od števila že zapisanih znakov:

BUM_(4,11)!

No, pravi algoritem LZ77 (Leta 1977 sta ga predlagala Lempel in Ziv), ki ga uporablja tudi program ZIP, vedno zapisuje trojčke (odmik, dolžina, naslednji znak). Zapis

GORI_NA_GORI_GORI.

pretvori v obliko

(0, 0, G)(0, 0, O)(0, 0, R)(0, 0, I)(0, 0, _)(0, 0, N)(0, 0, A)(3, 1, G)(8, 4, G)(5, 3, .)

Podobno koti pri faksiranju sporočil algoritem odmike in dolžine kodira s Huffmanovim algoritmom.

Algoritem LZW, imenovan po avtorjih Lempelu, Zivu in Welchu, namesto iskanja enakih vzorcev v nizu besedila sproti gradi slovar. Začne z osnovnim slovarjem in ga dopolnjuje z vsakim nizom, ki se pojavi v besedilu, v slovarju pa ga še ni. Poglejmo primer na besedilu

GORI_NA_GORI_GORI.

Vzemimo, da je osnovni slovar sestavljen samo iz znakov: A, G, I, N, O, R, _, . Vsakemu znaku priredimo številčno vrednost, na primer tako, kot je to prikazano v spodnji levi tabeli.

indeks	vpis
1	A
2	G
3	I
4	N
5	O
6	R
7	_
8	.

indeks	vpis
9	GO
10	OR
11	RI
12	I_
13	_N
14	NA
15	A_
16	_G
17	GOR
18	RI_
19	_GO
20	ORI
21	I.

Naša naloga je, da poiščemo najdaljši podniz, ki je že v slovarju. Poglejmo na našem primeru. Pri sprehodu čez besedilo opazimo, da imamo znak G že v slovarju, zato poskusimo še z nizom GO. Tega v slovarju ni, zato znaku G priredimo vrednost 2, niz GO pa dodamo v slovar pod naslednjo zaporedno številko (zgornja desna tabela). Iskanje nadaljujemo na enak način z znakom O. Ko se sprehodimo čez celotno besedilo, dobimo naslednji zapis:

G	O	R	I	_	N	A	_	G O	R I	_ G	O R	I	.
2	5	6	3	7	4	1	7	9	11	16	10	3	8

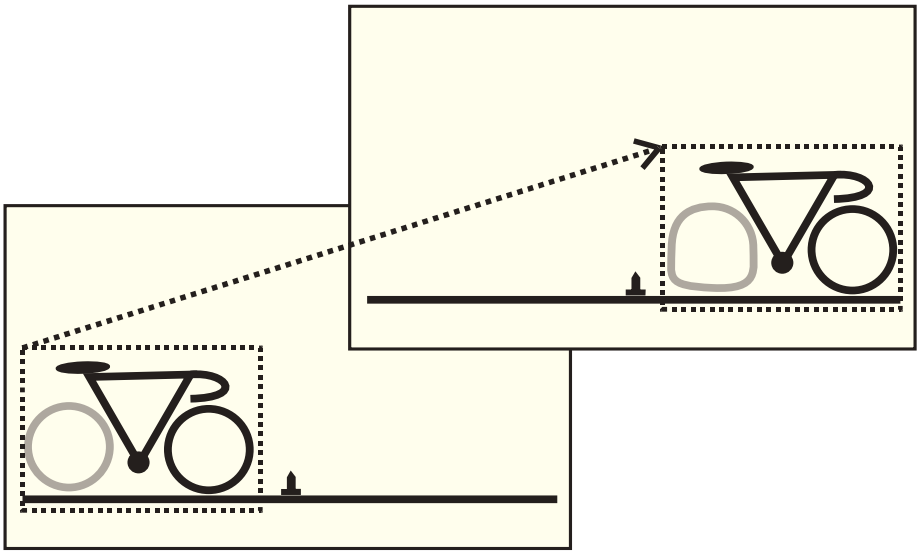
Dekodiranje tudi začnemo z osnovnim slovarjem, ki ga potem po vsakem sprejetem znaku dopolnimo. Ko sprejmemo 2, zapišemo G, ko sprejmemo 5, zapišemo O. Hkrati niz GO zapišemo v slovar. Potem sprejmemo 6, ki jo nadomestimo z R, v slovar pa vpišemo OR. Na ta način nadaljujemo do konca.

Huda zvijača: v slovarju imamo zapisani črki A in B z indeksoma 1 in 2. Kateri niz predstavljajo kode 1, 2, 3, 5?[1]

Stiskanje videa

Človeško oko lahko obdela deset do dvanajst slik na sekundo. Če ga izpostavimo hitrejšemu spreminjanju slik, ustvarimo navidezno gibanje. Na tej prevari sloni vsa filmska industrija. Danes se v filmih zamenja vsaj 24 slik vsako sekundo. Zaporedne slike so si zato med seboj zelo podobne. To lahko izkoristimo tudi pri stiskanju. Namesto, da bi stiskali vsako sliko posebej, lahko primerjamo dve zaporedni sliki. Enostavno samo povemo, v čem se nova slika razlikuje od prejšnje. Ta postopek izkoriščajo vsi moderni zapisi filmov (kodeki), na primer MPEG.

[1] ABABABA



Na zgornji sliki je dovolj, da povemo, za koliko se je spremenil položaj okvira, ki je orisan okrog kolesa in kako se je spremenila slika kolesa.

Nekaj podobnega počnejo tudi Bobri v nalogi Opisovanje filmov.

Varno kodiranje

Pri prenašanju in shranjevanju podatkov lahko pride do napak. Največkrat je vzrok v elektronskih vezjih, ki podležejo vplivom temperature ali zunanjih motenj. Pomislite na telefoniranje v mirnem in hrupnem okolju. Pogovor v hrupnem okolju bo potekal bistveno težje. Napake se lahko pojavijo kjerkoli in kadarkoli (prenosi podatkov, branje iz opraskane zgoščenke), vendar se je do neke mere mogoče obvarovati pred njimi ali jih celo popraviti. Ena možnost je, da izboljšamo zanesljivost fizičnega medija (tišje okolje pri telefoniranju), druga pa je uporaba shem za preverjanje pravilnosti podatkov, oziroma celo takih, ki znajo napake popravljati. Spet bomo gradili kode, le da tokrat poudarek ne bo na čim krajšem zapisu, temveč na povečevanju zanesljivosti v zameno za počasnejši prenos.

Če imamo v osnovni abecedi dva znaka, recimo A in B, in kodiramo binarno, enostavno priredimo enemu znaku eno vrednost, drugemu pa drugo, na primer 0 priredimo znaku A, 1 pa znaku B. Niz znakov AABBAB bi v tem primeru poslali kot 001101. Pri prenosu po žici ničlo največkrat predstavimo z nizko napetostjo, enico pa z visoko napetostjo. Zaradi zunanjih motenj, mogoče magnetnega polja, se lahko zgodi, da visoka napetost pade pod dovoljeno mejo. Na drugi strani bi potem namesto enice dobili ničlo. Če spet pogledamo naš primer – pri prenašanju tretjega znaka je prišlo do motenj in namesto 001101 smo na drugi strani sprejeli 000101. Rekonstrukcija nam prinese sporočilo AAABAB, ki je drugačno od poslanega.

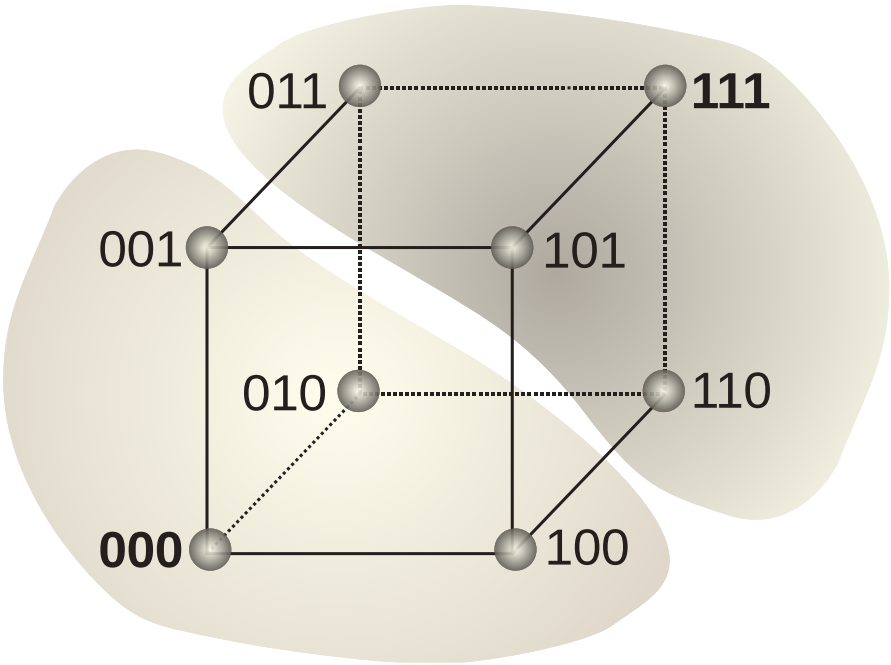
Ponavljajoče kode

Pomislimo, kako se pogovarjamo po telefonu v hrupnem okolju. Po potrebi prosimo sogovornika da ponovi izrečene besede. Na enak način lahko zagotovimo večjo jasnost tudi z računalnikom. Seveda je najbolj enostavno, da se računalniki ne dogovarjajo o ponovnem pošiljanju, ampak isti znak vsakič pošljejo večkrat. Namesto 0 in 1 lahko pošiljamo 00 in 11. Kodna zamenjava sedaj postane bistveno daljša. Za zgornji primer

AABBAB → 000011110011

vendar bistveno bolj zanesljiva. Kadar prejmemo 00 ali 11, vemo, da gre za znak A oziroma B, v primeru, da prejmemo 01 ali 10, pa vemo, da je prišlo pri prenosu do napake. Žal napake ne znamo popraviti. V zoprni situaciji, ko se narobe preneseta dva zaporedna znaka, pa naš sistem napake ne zazna. V primeru, da bi se zgornji niz prenesel kot 000000110011, bi narobe rekonstruirali osnovno sporočilo v AAABAB.

Več kot dodamo varnostnih znakov, bolje je. Pri uporabi treh binarnih simbolov za osnovni znak – A zamenjamo z 000 in B z 111 – je prenos bolj zanesljiv in seveda še počasnejši. Zdaj znamo napake celo popraviti. Če prejmemo več ničel kot enic, lahko sklepamo, da je bil poslan znak A, če je več enic kot ničel, pa znak B. Narišimo vse možne kombinacije v obliki grafa.



Edini pravi kodni zamenjavi sta odebeljeni. V okolici vsake od njiju pa so še tri kodne zamenjave, ki imajo vsaka po eno napako. Dobili smo dva otoka kodnih zamenjav, ki se med seboj nikjer ne prekrivata, zato lahko napake tudi popravljamo. Postopek pa seveda ni kos vsem napakam; če pride do dveh napak, recimo, da se 000 med prenosom spremeni v 101, pa znak ponovno narobe dekodiramo. Tu igramo na srečo, pri čemer se zavedamo, da so dvojne napake veliko manj verjetne od posameznih. Pa še posamezne napake naj bi se zgodile zelo malokrat.

Preverjanje sodosti

Na prej omenjeni način se dobro zavarujemo pred napakami, ni nam pa všeč, da prenos tako upočasnimo. V praksi se več uporabljajo kodi, ki na bolj zvite načine preverjajo skladnost podatkov. Velikokrat podatke razvrstimo v pravokotnik. Vzemimo niz znakov

ABBABABAABBABBA .

Znake osnovne abecede zamenjamo z ničlami in enicami (A → 0, B → 1) in jih zapišemo v obliki pravokotnika. Zapis predstavlja sivi pravokotnik na spodnji sliki

0	1	1	0	1
0	1	0	0	1
1	0	1	1	0

Za varnost lahko na koncu vsake vrstice in vsakega stolpca dodamo še en znak kodirne abecede tako, da je vsota v vsaki vrstici ali v vsakem stolpcu sodo število. Rezultat je v spodnji tabeli.

0	1	1	0	1	1
0	1	0	0	1	0
1	0	1	1	0	1
1	0	0	1	0	0

Ta koda lahko popravi eno napako, ne glede na katerem od 24 bitov se pojavi. Poglejmo nekaj primerov. Če je napaka v drugi vrstici in tretjem stolpcu, potem vsota v drugi vrstici in tretjem stolpcu ne bo dala sodega števila. Vrstica in stolpec z napako nam torej natančno določata bit, ki se je napačno prenesel. Nič drugače ni, če se je narobe prenesel kateri od varnostnih bitov.

Če se spremeni več bitov, vemo, da je nekaj narobe, ne znamo pa natančno ugotoviti kaj. Poglejmo, kaj se zgodi, če sta narobe prenesena prvi in drugi bit. Tabela je potem

1	0	1	0	1	1
0	1	0	0	1	0
1	0	1	1	0	1
1	0	0	1	0	0

Hitro vidimo, da je nekaj narobe v prvem in drugem stolpcu, ne vemo pa, v kateri vrstici. Napake ne moremo odpraviti.

Kaj pa kontrola lihosti? Ni problema, dokler je število obeh, stolpcev in vrstic, sodo ali liho število. Če temu ni tako, imamo težave s spodnjim desnim bitom.

V zgornjem primeru smo petnajst bitov, ki predstavljajo osnovni niz, varovali z devetimi varnostnimi biti. To je veliko bolje kot prej, ko smo vsak znak v osnovnem nizu varovali z dvema varnostnima bitoma. Prenosi so zato seveda hitrejši. Zato pa smo plačali tudi ceno. Prej smo znali popraviti eno napako v vsakem trojčku bitov, zdaj znamo zanesljivo popraviti eno napako v skupini 24 bitov.

Podobno shemo se da zasnovati tudi v obliki trikotnika. Spet vzemimo naš osnovni niz znakov ABBABABAABBABBA in ga z ničlami in enicami zapišimo v trikotnik

0	1	1	0	1
0	1	0	0	
1	1	0		
1	1			
0				

ter ga dopolnimo z biti za preverjanje sodosti. Zdaj bite nastavljamo tako, da je skupna vsota bitov v vrstici in stolpcu, v katerih je varnostni bit, soda:

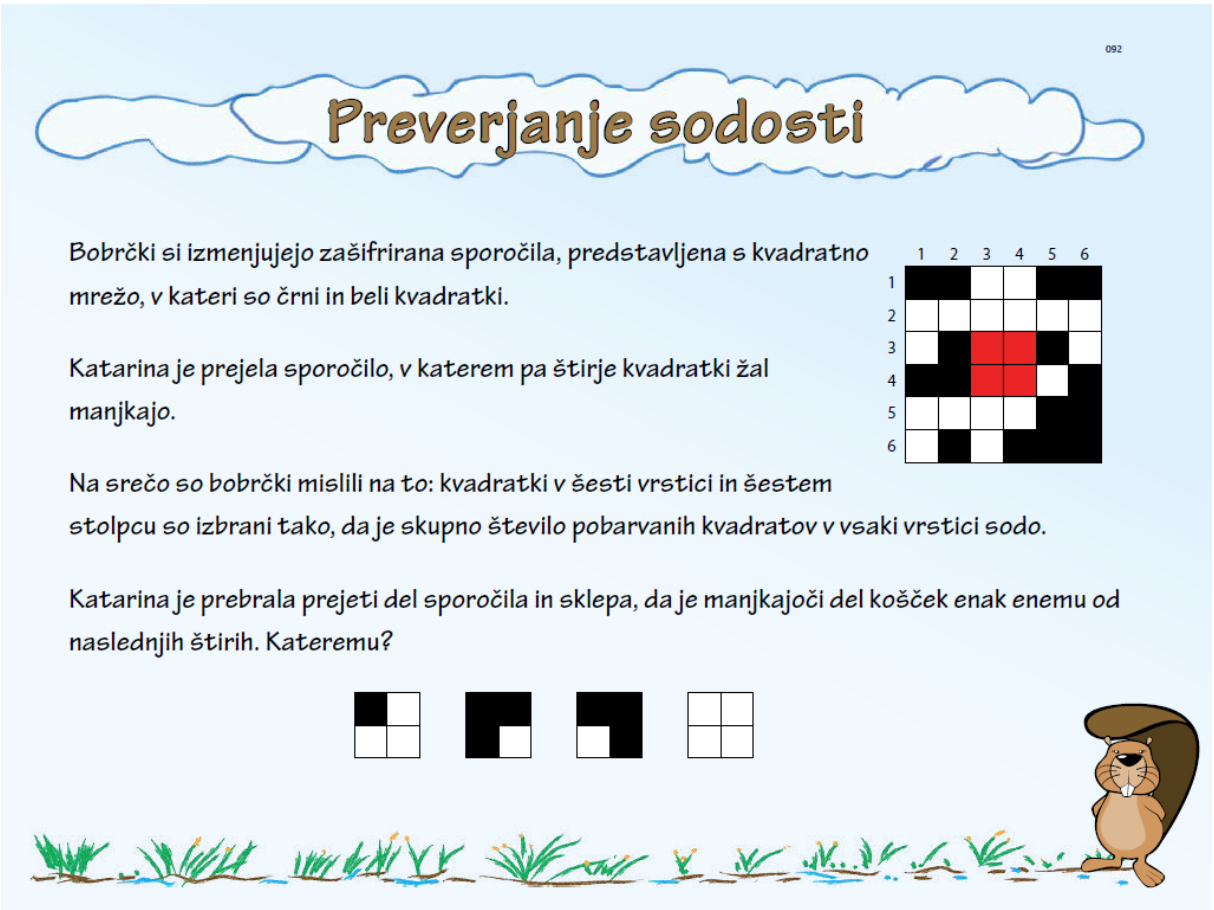
0	1	1	0	1	1
0	1	0	0	0	
1	1	0	0		
1	1	1			
0	0				
0					

Na zgornji sliki so biti, ki jih vključimo v računanje varnostnega bita v četrti vrstici in tretjem stolpcu, uokvirjeni. Tudi s tem kodom lahko odkrijemo in popravimo eno napako. Če je napaka na enem od bitov, ki predstavlja znak osnovnega niza, bosta napačna dva varnostna bita. Brez težav ugotovimo, za kateri bit gre. Če se narobe prenese varnostni bit, bo napaka ena sama. Vemo, da moramo obrniti ta bit.

Ta kod varuje petnajst bitov, ki predstavljajo osnovni niz, s šestimi varnostnimi biti. Pri popravljanju je enako zmogljiv kot pravokotnik kod, ker ima manj varnostnih bitov, zna odkriti manj dvojnih napak.

Z varnostnim kodiranjem ne moremo zagotoviti popolnoma varnega prenosa podatkov, lahko pa močno zmanjšamo možnost, da napake pri prenosu ne zaznamo. Kakšno varnostno kodiranje uporabiti, je močno odvisno od zahtev vsakega sistema posebej.

S preštevanjem kvadratkov se spopadajo tudi bobri v nalogi Preverjanje sodosti.



Varno kodiranje v vsakdanjem življenju

Po analogiji s preverjanjem parnosti lahko preverjamo tudi pravilnost zapisov številk, recimo bančnih računov, kod ISBN, črtnih kod in osebnih identifikacijskih številk.

Koda ISBN natančno določa knjigo. Koda ima od 10 do 13 števk. Poglejmo zapis z deset števkami:

0-691-12418-3 .

Prva števka določa jezik (0 za angleščino), naslednje dve ali tri pa določajo založnika (691 je Princeton University Press). Naslednjih pet števk določa oznako knjige, ki jo dodeli založnik. Zadnja številka je namenjena preverjanju pravilnosti kode in lahko zavzame vrednosti od 0 do 10. Če je zadnja vrednost 10, jo zamenjamo s črko X. Zadnji znak mora biti določen tako, da velja

$$z_1 + 2z_2 + 3z_3 + 4z_4 + 5z_5 + 6z_6 + 7z_7 + 8z_8 + 9z_9 + 10z_{10} = 0 \bmod 11 \quad .$$

Z z_1 do z_{10} so označeni znaki v kodi. Za naš primer lahko preverimo

$$0 + 2 \times 6 + 3 \times 9 + 4 \times 1 + 5 \times 1 + 6 \times 2 + 7 \times 4 + 8 \times 1 + 9 \times 8 + 10 \times 3 = 198 = 11 \times 18 = 0 \bmod 11$$

Koda je zasnovana tako, da zna odpraviti eno narobe zapisano števko in zamenjavo dveh zaporednih števk. Gre za dve najbolj pogosti napaki pri tipkanju.

V Sloveniji za identifikacijo oseb uporabljamo številko EMŠO. EMŠO je sestavljen iz 13 števk, recimo:

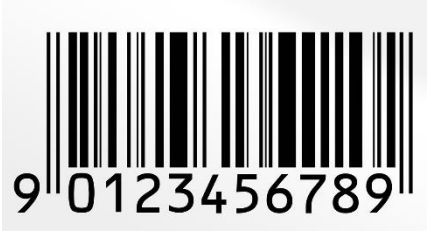
2809013505005 .

V zgornji kodi 28 označuje datum rojstva, 09 mesec rojstva, 013 leto rojstva brez tisočic, 50 Slovenijo (koda EMŠO je z nami še od časov Jugoslavije, druge republike so uporabljale druge številke območja), naslednje tri števke pa določajo spol in zaporedno številko rojstva na ta dan. Za moške zavzamejo vrednosti od 000 – 499, za ženske pa od 500 – 999. Zadnja števka je kontrolna. Določimo jo tako, da je vsota, ki jo dobimo po spodnjem receptu, deljiva z 11:

	2	8	0	9	0	1	3	5	0	5	0	0	5	
x	7	6	5	4	3	2	7	6	5	4	3	2	1	
	14	48	0	36	0	2	21	30	0	20	0	0	5	= 176

V našem primeru je ustrezna kontrolna številka 5, saj je 176/11 = 16. V primeru, da bi morali uporabiti kontrolno številko 10, zaporedno številko povečamo za 1 in ponovimo postopek.

Na izdelkih v trgovini najdemo črtno kodo. Obstaja množica različnih črtnih kod. Zaradi lažjega branja kod z elektronskimi bralniki je vsaka številka, napisana pod kodo, predstavljena z več debelimi ter tankimi črtami in presledki med njimi. Kadar je koda zmečkana ali pa slabo odtisnjena, elektronski bralnik zelo težko pravilno prebere vse števke. V takem primeru kontrolna vsota ne drži in prodajalka mora kodo vtipkati ročno. Za nekatere sisteme črtnih kod je način preverjanja standardiziran, lahko pa si ga zamislimo popolnoma po svoje.


9 0123456789

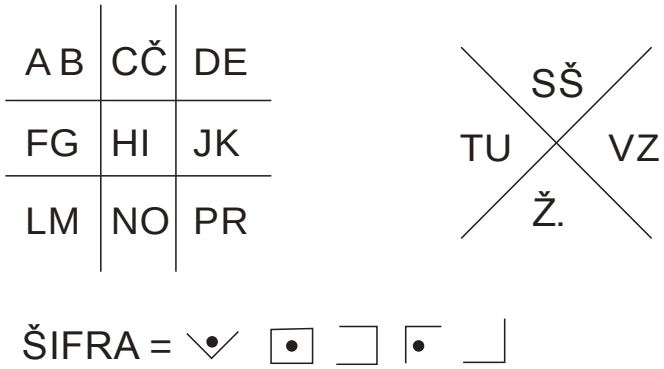
Šifriranje

Skrivna komunikacija je verjetno stara toliko kot človeštvo. Prve znane zapisane šifre so odkrili v Egiptu in so stare več kot štiri tisoč let. Skozi tisočletja so se razvili mnogi spodobni šifrirni sistemi. Njihova zapletenost je s časom naraščala, tako kot se je bogatilo znanje in sposobnost, da ljudje z najrazličnejšimi pripomočki šifre razbijemo. Danes se v elektronskih komunikacijah šifrirni sistemi intenzivno uporabljajo. Seveda so mnogo bolj kompleksni od teh, ki si jih bomo ogledali v nadaljevanju. Slonijo pa na enakih idejah.

Osnovno besedilo ali čistopis po nekem pravilu preoblikujemo v tajnopis. Poleg pravila postopek preoblikovanja določa tudi ključ. Pri šifriranju vedno predpostavimo, da je postopek šifriranja znan (prej ali slej se razve), razen pošiljatelja in prejemnika pa naj nihče ne bi poznal ključa.

Šifriranje z alternativno abecedo

Ena najbolj znanih alternativnih abeced je prostozidarska ali skavtska abeceda. Pravilo, po katerem se črke slovenske abecede preslikajo v črke prostozidarske abecede, je prikazano na spodnji sliki.



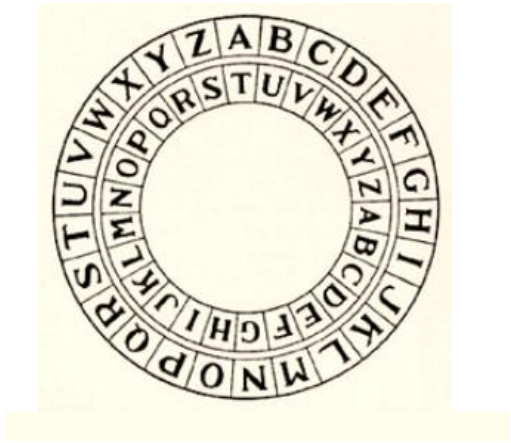
Črke razvrstimo v dve tabeli, kot kaže slika. Pri šifriranju posamezno črko nadomestimo s črtama, ki črko ločijo od ostalih črk, z izjemo sosede. Sosednji črki ločimo tako, da prvi narišemo samo črte, drugi pa dodamo še piko.

To šifro je zelo enostavno razbiti. Preštejemo pogostost posameznih znakov in jo primerjamo s pogostostjo znakov v običajnih besedilih. Najpogostejše znake bomo hitro izluščili, manj pogoste pa bomo lahko dobili iz besednih zvez. Daljše kot bo zašifrirano besedilo, bolj natančno bomo lahko rekonstruirali kodno tabelo.

Šifriranje z zamenjavami

Izmišljanje nove abecede je precej nerodno. Veliko raje pišemo znake, ki smo jih navajeni. Šifrirni sistem z zamenjavami so si zamislili že v starem Rimu v času Julija Cezarja. Po njem se tudi imenuje. Ideja je silno preprosta. Znake osnovne abecede zamenjamo z drugimi znaki

abecede, pri tem pa vrstnega reda znakov ne spremenimo. Rimljani so uporabljali šifrirno kolo



šifriramo pa lahko tudi s tabelo. V nadaljevanju bomo čistopis pisali z malimi črkami, tajnopis pa z velikimi. V spodnjem primeru je šifrirna abeceda zamaknjena za tri znake glede na osnovno abecedo. Ključ je torej 3 ali pa črka Č, s katero se začne šifrirna abeceda.

a	b	c	č	d	e	f	g	h	i	j	k	l	m	n	o	p	r	s	š	t	u	v	z	ž
Č	D	E	F	G	H	I	J	K	L	M	N	O	P	R	S	Š	T	U	V	Z	Ž	A	B	C

Šifriranje poteka enostavno. Znak iz čistopisa poiščemo v zgornji vrsti in namesto njega zapišemo znak tajnopisa iz spodnje vrste. Primer:

cezarjeva šifra → EHBČTMHAČ VLITČ

To šifro je zelo enostavno razbiti. V tajnopisu poiščemo najbolj pogosti znak. Najbolj pogosta črka v slovenščini je e, sledijo ji a, o in i. Zelo verjetno je da najbolj pogost znak v tajnopisu zamenjuje črko e. Kaj skriva tajnopis

ŽUŠHOS RČP MH ?

Boljšo šifro bi dobili, če bi črke poljubno premešali. V tem primeru si je ključ težje zapomniti. Pred slabimi 500 leti je Vigenere nadgradil Cezarjevo šifro tako, da je hkrati uporabil več Cezarjevih šifrirnih abeced. Ključ pri Vigenerejevi šifri je navadno kar beseda ali besedna zveza, recimo KRT. Če pri roki nimamo tablice z vsemi možnimi Cezarjevimi šifrirnimi abecedami, si pripravimo samo nujno potrebne – to so abecede, ki se začnejo s črkami K, R in T:

a	b	c	č	d	e	f	g	h	i	j	k	l	m	n	o	p	r	s	š	t	u	v	z	ž
K	L	M	N	O	P	R	S	Š	T	U	V	Z	Ž	A	B	C	Č	D	E	F	G	H	I	J
R	S	Š	T	U	V	Z	Ž	A	B	C	Č	D	E	F	G	H	I	J	K	L	M	N	O	P
T	U	V	Z	Ž	A	B	C	Č	D	E	F	G	H	I	J	K	L	M	N	O	P	R	S	Š

Z njo bomo zašifrirali zgoraj nerazkriti čistopis. Pod čistopis podpišemo ključ. Če je ključ prekratek, ga ponovimo. Črka ključa nam določa, katero Cezarjevo šifrirno abecedo uporabimo za šifriranje znaka čistopisa:

u	s	p	e	l	o		n	a	m		j	e
K	R	T	K	R	T		K	R	T		K	R
G	J	K	P	D	J		A	R	H		U	V

Zdaj so stvari bolj zapletene. Vigenerejev sistem je porušil strukturo besed. Hitro opazimo, da se ista črka čistopisa preslika v različne črke v tajnopisu. Obratno, ista črka tajnopisa lahko predstavlja različne črke v čistopisu. Šifro spet napademo s frekvenčno analizo, ki jo moramo delati za vsako črko ključa (Cezarjeve šifrirne abecede) posebej. Pri tem seveda ne vemo, kako dolg je ključ.

Šifriranje s premeščanjem

Gre za eno bolj enostavnih šifer. Po nekem pravilu znake premešamo med seboj. Dokler se je šifriralo in dešifriralo na roke, je bila šifra precej v uporabi. Zašifrirajmo sporočilo

antična šifra .

Temu sporočilu pravimo tudi čistopis. Pri premeščanju se čistopis največkrat napiše v matriko po vrsticah:

a n t i
č n a š
i f r a

preberemo pa ga po stolpcih:

AČINNFTARIŠA .

Šifro lahko dodatno zapletemo, če stolpce premešamo med seboj, preberemo po diagonalah, v obliki spirale in podobno. To šifro je precej težko razbiti na roke. Ne moremo se zanašati na pogostost posameznih znakov, ampak moramo analizirati pogostost parov znakov. Za določitev števila znakov v vrstici pa nam prav pa pride tudi opazovanje razmerja med samoglasniki in soglasniki.

Šifriranje s premeščanjem so poznali že Špartanci. Čistopis so napisali na trak, trak ovili okrog palice in nato vsebino prebrali vzdolž palice.

Šifriranje z javnim in zasebnim ključem

Teorija pravi, da z daljšanjem ključa dobimo bolj varne tajnopise. Če je ključ dolg toliko kot čistopis, je šifrirni sistem popolnoma varen. Tudi Vigenerejev. To pa nas pripelje do še enega

problema. Ključ morata poznati pošiljatelj in sprejemnik. Daljši kot je, težje si ga izmenjata in bolj nevarno je, da ga kdo prestreže.

Za izmenjevanje ključev se danes uporabljajo šifrirni sistemi z javnim in zasebnim ključem. Ti temeljijo na kompleksnosti postopkov - dešifriranje je mnogo bolj zapleteno od samega šifriranja. Danes se največ uporabljajo postopki, ki temeljijo na praštevilih. Iz dveh gromozanskih praštevil, ki jih računalniki dokaj hitro izračunajo, dobimo še večji produkt. Ta produkt je zelo težko razcepiti na obe praštevili in s tem rabiti šifro.

Danes se pri varni elektronski komunikaciji omenjajo certifikati. Gre za sistem javnih in zasebnih ključev, za katerimi se navadno skriva šifrirni sistem RSA. Ta je zasnovan ravno okrog ideje z velikimi praštevili. Javni ključ vsebuje samo produkt praštevil, ki ga je tako rekoč nemogoče razbiti na prafaktorja. Zasebni del ključa pa vsebuje prafaktorja. Šifriranje je zasnovano tako, da z javnim ključem lahko kdorkoli šifrira sporočila, odšifrira pa jih lahko samo lastnik, ki pozna prafaktorja. Na žalost pa je postopek šifriranja in dešifriranja (ob znanih prafaktorjih) preveč počasen. Zato se na ta način vedno prenesejo šifrirni ključi, nadaljevanje šifrirane komunikacije pa poteka po hitrih postopkih z enim samim ključem.

Bobri se v nalogah Cocsozšla tigsa in Golobi z dolgim nosom spopadajo s šifrirnimi sistemi z enim samim ključem. Znamo ugotoviti za katere šifrirne sisteme gre?

Golobi z dolgim nosom

Bober Matjaž pošilja prijateljici, bobrovki Alenki, razglednico s počitnic. Ker poštni golobi radi povsod vtikajo nosove, bo sporočilo skril. Napisal ji je

ŽAJTAMJOVTAJLIŠOPITAJROMZEVARDZOPEPEL

kar pomeni: Lepe pozdrave z morja ti pošilja tvoj Matjaž

Alenka mu je odgovorila

AKNELAOCITRAKAZALAVH

Kaj to pomeni?